Lowent Manual

Systems Programming with Contracts and Effects

draft

v0.2.0 · last updated 2026-09-15

rubidus

rubidus@gmail.com github.com/rubidus-api/lowent_lang

An introduction to the Lowent language, and its user manual.

Written for readers who have programmed a little,
up to those who already use a systems language such as C or Rust.

Copyright and contact

author	rubidus
contact	rubidus@gmail.com
repository	github.com/rubidus-api/lowent_lang
edition	v0.2.0 — draft
last updated	2026-09-15

The text — Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0). You may share and adapt it freely so long as you credit the source; commercial use is not permitted, and adaptations must carry the same licence.
https://creativecommons.org/licenses/by-nc-sa/4.0/

The example code — the MIT licence. Take it and use it freely. The Lowent compiler and standard library used by the examples are under the repository's MIT licence.

Every demonstration in this book prints output really obtained by checking and running the code with `lowentc`; programs that run were also checked to give the same answer on the VM and in a native build. The typesetting is done with Typst.

Lowent is an **experimental** language. Its syntax and library may change, and this book is revised along with each edition.

This book keeps being corrected. What you are reading is the edition numbered above; corrections and additions follow it. The newest edition and the record of changes are on the author's GitHub. If you are holding an older copy, look there first. Error reports and suggestions are received in the same place.

A note on this translation

The Korean edition is the original. This English edition is translated from it chapter by chapter, and **chapter numbers are kept identical to the original**, so a cross-reference to "chapter 12" means the same chapter in both editions. The examples are shared: both editions print the same programs and the same verified output.

Contents

A note on this translation 2

Preface 16
- What this book promises and does not promise 16
- Who this book is for 16
- Editions and licence 17

How to read this book 17
- Path ① — learning this language from the start 17
- Path ② — familiar with Rust or C and wanting only the differences 17
- Path ③ — writing for machines without an operating system 17
- Path ④ — wanting to know how far this language's claims are true 17
- Devices in the text 17

Part I — Getting started

1. What Lowent sets out to do 20
   - 1.1 Can you tell from the head alone? 20
   - 1.2 Five ideas 22
   - 1.3 One meaning, one spelling 23
   - 1.4 Where the language stands 23
   - 1.5 How to read this book 24

2. A first program — build, run, get rejected 25
   - 2.1 Building the tool 26
   - 2.2 hello, entropy 27
   - 2.3 Running it two ways 28
   - 2.4 How a contract stops execution 28
   - 2.5 Getting rejected 29
   - 2.6 `--fmt` fixes the shape 30
   - 2.7 Arguments and capabilities 31
   - 2.8 Common mistakes 31
   - 2.9 This chapter's syntax at a glance 34

3. The surface — full stops, blocks and clause order 35
   - 3.1 The name first, then the arguments 36
   - 3.2 A free-standing full stop closes 36
   - 3.3 A block is always `do … end` 38
   - 3.4 Comments and literals 39
   - 3.5 Names 42
   - 3.6 Words are a budget 42
   - 3.7 The boundaries of the `expr` island 43
   - 3.8 Clause order in an op head 43
   - 3.9 Common mistakes 45
   - 3.10 This chapter's syntax at a glance 48

Part II — Values and flow

4. Numbers — fixed-width integers and floating point 51
   - 4.1 Integer types name their width 52
   - 4.2 Widening is automatic, narrowing is written 52
   - 4.3 Overflow stops 54
   - 4.4 Division and remainder 55

4.5 Booleans are not numbers ... 56
4.6 Bitwise operations ... 57
4.7 Floating point ... 58
4.8 Common mistakes ... 60
4.9 This chapter's syntax at a glance ... 65

5. Ops — `fn` and `proc` ... 66
5.1 Two kinds ... 67
5.2 Purity is decided by observation ... 68
5.3 Don't write `effects` on a `fn` ... 69
5.4 Effects spread to the caller ... 70
5.5 Parameters and return ... 70
5.6 Only `neg` is unary ... 71
5.7 Modifiers ... 72
5.8 Common mistakes ... 72
5.9 This chapter's syntax at a glance ... 75

6. Locals — `let` and `var` ... 76
6.1 The word says whether it changes ... 77
6.2 Write the type, or leave it to the value ... 78
6.3 No name without a value ... 79
6.4 How long a local lives ... 79
6.5 `if` yields no value ... 81
6.6 Common mistakes ... 81
6.7 This chapter's syntax at a glance ... 84

7. Flow — branches, loops and leaving early ... 85
7.1 Conditions and loops ... 86
7.2 `guard` — turning a condition into a fact ... 87
7.3 Every path returns a value ... 89
7.4 `match` — every case, none missing ... 90
7.5 `panic` is an effect ... 91
7.6 Common mistakes ... 92
7.7 This chapter's syntax at a glance ... 96

8. Expressions — prefix notation and the `expr` island ... 97
8.1 Nothing to memorise in prefix notation ... 98
8.2 The island's precedence table ... 99
8.3 What the island cannot do ... 100
8.4 Short-circuiting guards conditions ... 101
8.5 Expressions computed at translation time ... 102
8.6 Common mistakes ... 103
8.7 This chapter's syntax at a glance ... 106

Part III — Data

9. Sequences — arrays and slices ... 109
9.1 Start and length together ... 110
9.2 Writing needs `mut slice` ... 111
9.3 Narrowing the window ... 112
9.4 Contracts remove bounds checks ... 114
9.5 Common mistakes ... 116
9.6 This chapter's syntax at a glance ... 119

10. Aggregates — `struct` and `enum` ... 120

10.1 `struct` — a collection of named fields 121
10.2 `enum` — one of several 122
10.3 Variants are closed with full stops 124
10.4 Nothing may contain itself 124
10.5 Common mistakes 126
10.6 This chapter's syntax at a glance 129

11. Types that hold answers — `option` and `result` 130
11.1 `option` — a value, or none 131
11.2 `result` and the `errors` clause 133
11.3 `try` — passing failure upwards 135
11.4 Crossing between the two channels 135
11.5 Three ways to report failure 136
11.6 Combining and nesting patterns 137
11.7 Common mistakes 139
11.8 This chapter's syntax at a glance 143

12. Borrowing — `ref` and `mut_ref` 144
12.1 Two kinds of borrow 145
12.2 Write permission only narrows 146
12.3 Many readers or one writer 147
12.4 A borrow cannot outlive what it borrows 148
12.5 There is no `mut ref slice` 149
12.6 Common mistakes 150
12.7 This chapter's syntax at a glance 153

13. Named types — `type`, `newtype`, `range`, `cast` 154
13.1 `type` is an alias, `newtype` a new type 155
13.2 `range` — a contract that became the shape of a parameter 158
13.3 `cast` — where a value may change 159
13.4 `bits` — from 1 to 64 bits 160
13.5 Pinning down byte layout 160
13.6 Both directions of a layout, and a view that can fail 161
13.7 `bitset` — a set of small numbers 162
13.8 Scattered pieces as one view — `segments` 164
13.9 Three kinds of type words 165
13.10 Common mistakes 166
13.11 This chapter's syntax at a glance 169

Part IV — Contracts and effects

14. Contracts — write them, have them checked, lose the checks 172
14.1 A contract is a checked promise 173
14.2 Whose fault is it? 173
14.3 Contracts remove checks 175
14.4 Conditions over every element 175
14.5 Naming a contract 176
14.6 Decide now what can be decided now 177
14.7 Contract grades 178
14.8 Build modes decide the remaining checks 179
14.9 Common mistakes 180
14.10 This chapter's syntax at a glance 183

15. Effects — the marks an op leaves on the world 184

15.1 Effects are closed words ... 185
15.2 Effects spread to the caller ... 186
15.3 Declared but not done is reported ... 187
15.4 The effects line is a set ... 188
15.5 `via` — a type argument decides the effects ... 189
15.6 What purity allows ... 189
15.7 Common mistakes ... 190
15.8 This chapter's syntax at a glance ... 194

16. Capabilities — power that is handed over ... 195
16.1 Kinds of capability ... 196
16.2 Capabilities travel down the chain ... 196
16.3 Effects and capabilities are a pair ... 198
16.4 Kind, not presence ... 200
16.5 What the entry point receives ... 200
16.6 Common mistakes ... 202
16.7 This chapter's syntax at a glance ... 205

17. Designing failure ... 206
17.1 The boundary and the inside ... 207
17.2 How to split an error enum ... 208
17.3 Moving up the layers ... 209
17.4 Where information may be thrown away ... 210
17.5 Where `panic` is acceptable ... 211
17.6 Common mistakes ... 211
17.7 This chapter's syntax at a glance ... 215

Part V — Memory

18. Regions — where values live, and memory reclaimed all at once ... 218
18.1 Three places values live ... 219
18.2 Opening a region and obtaining space ... 220
18.3 Receiving a region ... 222
18.4 Nothing is carried out of a region ... 222
18.5 One cursor per root ... 223
18.6 The growing root ... 224
18.7 A stack on a region ... 225
18.8 Common mistakes ... 226
18.9 This chapter's syntax at a glance ... 231

19. Ownership — one party responsible for disposal ... 232
19.1 `owned` and `drop` ... 233
19.2 Where branches meet ... 235
19.3 Release and completion ... 236
19.4 Not counting on the operating system to clean up ... 238
19.5 At what strength is memory safety guaranteed? ... 238
19.6 Common mistakes ... 239
19.7 This chapter's syntax at a glance ... 242

20. Allocators and fixed memory ... 243
20.1 The three parts of an allocator ... 244
20.2 Cutting borrowed bytes ... 245
20.3 Swapping allocators ... 246
20.4 Default allocators that carve straight from a root ... 249

20.5 Growing and returning a piece — `grow` and `release` 250
20.6 Where the three layers sit in the standard library 252
20.7 Who sets the size of the fixed window? 252
20.8 Same bits, different reading 253
20.9 Common mistakes 254
20.10 This chapter's syntax at a glance 258

Part VI — Abstraction

21. Modules — hidden by default 261
21.1 One file, one module 262
21.2 Importing 263
21.3 What can be exported, and how to shorten a name 264
21.4 There is no search path 265
21.5 When names collide 266
21.6 Top-level order does not matter 266
21.7 Common mistakes 267
21.8 This chapter's syntax at a glance 272

22. Generics — parameters fixed at translation time 273
22.1 Taking values and types at translation time 274
22.2 An instance per combination 276
22.3 Putting conditions on types 276
22.4 The type carries it, not a value 278
22.5 Common mistakes 279
22.6 This chapter's syntax at a glance 283

23. Traits — one promise kept by many types 284
23.1 What they are for 285
23.2 Ops attached to types, and `method` 286
23.3 Traits with several ops 287
23.4 Signatures do not say `fn` or `proc` 288
23.5 When a trait is not satisfied 290
23.6 `via self` — more allocation effects only 291
23.7 Common mistakes 293
23.8 This chapter's syntax at a glance 296

24. `pipe` — one line for each thing you mean to do 297
24.1 The same work, two shapes 298
24.2 Stages and terminals 299
24.3 Walking once is the definition 304
24.4 Reading only as much as needed 305
24.5 The built-in `map` · `filter` with the same names 305
24.6 Common mistakes 306
24.7 This chapter's syntax at a glance 310

Part VII — Concurrency

25. Actors — living with state, by messages 312
25.1 Declare, make, talk 313
25.2 Carrying values in messages 314
25.3 Mailboxes — putting in and emptying 316
25.4 Let it crash, then restart 317
25.5 Where actors may be used 319

25.6 Actors that hold a capability, and what state may contain 319
25.7 Designing with actors — a transfer between two accounts 322
25.8 Common mistakes 323
25.9 This chapter's syntax at a glance 330

26. Tasks and channels — exchange between bound flows 331
26.1 Flows are bound to blocks 332
26.2 Channels — containers with an order 334
26.3 Deadlocks you can see from what is written 336
26.4 Flows and memory 337
26.5 What does not exist yet 340
26.6 Common mistakes 341
26.7 This chapter's syntax at a glance 344

27. Parallel loops and atomic operations 345
27.1 Declaring that a loop may be split 346
27.2 The three conditions the processor checks 348
27.3 The combining operation must be associative 349
27.4 Atomic operations 350
27.5 Memory orderings 351
27.6 Lanes — computing several values at once 352
27.7 Saying how a place is used — `access` 356
27.8 Common mistakes 357
27.9 This chapter's syntax at a glance 361

Part VIII — The outside world

28. Input, output and files 364
28.1 Standard output 365
28.2 Reading a whole file 365
28.3 What is opened is closed 366
28.4 End and failure are different answers 368
28.5 Triggering failure on purpose 368
28.6 Common mistakes 369
28.7 This chapter's syntax at a glance 373

29. Meeting C 374
29.1 Calling C — mark, right, effects line 375
29.2 Types that cross the boundary 377
29.3 C calls Lowent 377
29.4 Callbacks and ownership 379
29.5 Common mistakes 380
29.6 This chapter's syntax at a glance 383

30. Hardware — registers, interrupts, machine instructions 384
30.1 The register map 385
30.2 Access is enforced at translation 387
30.3 Interrupt handlers 388
30.4 What a machine supports — `build tier` 389
30.5 Machine instructions — `asm` 390
30.6 The absorbing boundary — where `unsafe` stops 392
30.7 Using what the machine has — `--hw` 393
30.8 Common mistakes 393
30.9 This chapter's syntax at a glance 396

31. Building and testing — packages, configuration, tests, cross-checks 397
31.1 The manifest and the project 398
31.2 Dependencies are pinned by hash 399
31.3 A different program per build — `build option` and `config` 400
31.4 Tests 402
31.5 How the compiler is verified 404
31.6 Asking where it is slow 405
31.7 Common mistakes 405
31.8 This chapter's syntax at a glance 408

Part IX — The standard library

32. A map of the standard library 411
32.1 Three layers — language, leaf, library 412
32.2 What gets in 412
32.3 The map by layer 413
32.4 File names and module names 414
32.5 Conventions every module follows 415
32.6 The list of leaf ops — what lies beneath the modules 415
32.7 Common mistakes 418
32.8 This chapter's syntax at a glance 419

33. Text and encodings — `strings`, `fmt`, `utf8`, `codec`, `hash` 420
33.1 Cut, find, assemble 421
33.2 UTF-8 — no replacement characters 422
33.3 Hex and base64 423
33.4 Hashes — three questions, three answers 424
33.5 Regular expressions — no backtracking 425
33.6 Common mistakes 425
33.7 This chapter's syntax at a glance 428

34. Containers and sorting — `sortlib`, `sortgen`, `hashmap`, `vecgen`, `spsc` 429
34.1 Sorting and finding `u64` 430
34.2 The type brings the criterion 430
34.3 A hash map over the caller's slice 431
34.4 A growing generic vector 432
34.5 A ring buffer passing values between flows 433
34.6 Other containers 434
34.7 Common mistakes 434
34.8 This chapter's syntax at a glance 437

35. Storage and handles — `pool`, `shard`, `budget`, `wire` 438
35.1 Generational handles 439
35.2 Brands — no mixing pools 440
35.3 Non-overlapping pieces — `shard` 441
35.4 Bit budgets — `budget` 442
35.5 Bit fields — `wire` 443
35.6 Other storage modules 443
35.7 Common mistakes 444
35.8 This chapter's syntax at a glance 446

36. Input/output, networking, time, randomness, cryptography 447
36.1 Buffered output — forget to flush and translation refuses 448
36.2 Networking — handles are resources 449

36.3 Randomness — reproducible sequences and operating-system entropy 450
36.4 Time — monotonic clocks and wall clocks 451
36.5 The HTTP request parser — its heart is rejection 452
36.6 Cryptography — the order of the stack and the missing top 453
36.7 Common mistakes 453
36.8 This chapter's syntax at a glance 456

37. The terminal — `term` and `tty` 456
37.1 `term` does not write to the screen 457
37.2 Redrawing only changed cells 458
37.3 The number of cells is not the number of bytes 459
37.4 Reading keys is computation 460
37.5 Raw mode is a capability 461
37.6 What is not built yet 462
37.7 Common mistakes 462
37.8 This chapter's syntax at a glance 464

Part X — Grounds: what has been proven

38. Why prove 467
38.1 Tests show existence, proofs show absence 468
38.2 Overlapping three layers 469
38.3 What is proven 470
38.4 Three ways of proving 471
38.5 Promises this part keeps 471

39. The mathematical toolkit 472
39.1 Six tools 473
39.2 Sets and relations 474
39.3 Partial orders — incomparable pairs 474
39.4 Lattices and join — what to pick when combining 474
39.5 Monotone functions and fixed points — why analysis stops 475
39.6 Induction and invariants — dominoes 476
39.7 Abstract interpretation — ranges instead of values 477
39.8 Table of symbols 477
39.9 How the tools interlock 477

40. Proofs about numbers — widening, narrowing, division 478
40.1 Widening is a partial order 479
40.2 Division and remainder 481
40.3 Theorems that `range` provides 482
40.4 Putting stopping into the model honestly 483
40.5 What is not proven 483

41. Proofs about bounds — intervals, relations, row-major addresses 484
41.1 Computing with intervals 485
41.2 Relations — what intervals cannot do 486
41.3 Four stages 487
41.4 The row-major proof 489
41.5 Trust proofs, but back trust with checks 490
41.6 What is not proven 491

42. Proofs about ownership and borrowing 492
42.1 Two machines and three events 493
42.2 The central theorem 494

42.3 Shapes rejected and shapes accepted ... 495
42.4 Finding an implementation defect while widening the model ... 495
42.5 Between model and implementation ... 496
42.6 What is not proven ... 496

43. Loops and fixed points — proving “however many times” ... 497
43.1 It breaks on the second round ... 498
43.2 At a fixed point, any number of times ... 499
43.3 Only one dangerous shape ... 500
43.4 Fresh tokens — one move that removes the naming problem ... 500
43.5 Nested loops — from function to relation ... 501
43.6 What is not proven ... 502

44. Proofs about effects — writing “what it can do” into the type ... 502
44.1 Effects are sets of atoms ... 504
44.2 Four rules the processor enforces ... 504
44.3 Capabilities — the partner of effects ... 506
44.4 Effect soundness — not circular ... 506
44.5 Optimisations relying on effects are legitimate ... 507
44.6 What is not proven ... 508

45. Proofs about races and parallelism — discipline instead of a memory model ... 508
45.1 The Bernstein condition — an answer from 1966 ... 509
45.2 Independence means no races, and the condition is necessary ... 510
45.3 Between actors there is always a message ... 511
45.4 Containment — where heavy tools are needed ... 511
45.5 No races is not the same as deterministic ... 511
45.6 What is not proven ... 513

46. Proofs about weak memory — with the default, you may think sequentially ... 513
46.1 Sequential consistency and its counterexample ... 514
46.2 An execution is a graph ... 515
46.3 Impossible with the default, possible with `relaxed` ... 515
46.4 Exactly that axiom forbids it ... 516
46.5 Message passing — the weak ones exist for this shape ... 516
46.6 Over all executions ... 517
46.7 Weaker orderings add behaviours ... 517
46.8 Seven tests from the literature ... 518
46.9 What is not proven ... 518

47. Proofs about locks — where heavy tools are really needed ... 519
47.1 The vocabulary of separation logic ... 520
47.2 There cannot be two tokens ... 521
47.3 Tools only as needed ... 522
47.4 rwlock — two proofs of the same sentence ... 522
47.5 Deadlock freedom and no starvation ... 522
47.6 A borrowed proof in weak memory — `spsc` ... 523
47.7 What is not proven ... 525

48. Proofs about syntax — whichever closer closes it, the same tree ... 525
48.1 Algebra and denotation ... 526
48.2 One statement is a one-statement block ... 527
48.3 Closers do the same job ... 528
48.4 Common mistakes ... 530

48.5 What is not proven ... 531

49. Proofs about hashes — calling things by content, not name ... 531
49.1 Hash functions and Merkle DAGs ... 532
49.2 Called by content, not by name ... 533
49.3 What changes a hash and what does not ... 533
49.4 What can and cannot be proven ... 535
49.5 What hashes stop ... 536
49.6 What is not proven ... 537

50. What is not proven ... 537
50.1 The trusted base — what proofs rely on ... 538
50.2 The largest gap — between model and implementation ... 539
50.3 What remains — by topic ... 540
50.4 Not sold as “fully statically safe” ... 542
50.5 So what may you trust ... 542

Appendices and index

Appendix A — Words and builtins ... 545
The forty-three words ... 545
Removed words ... 545
Frequently used builtins ... 546
Effects and capabilities ... 547

Appendix B — Diagnostics index ... 547
At translation time ... 547
When execution stops ... 554
Other diagnostics ... 554

Appendix C — Common mistakes and how to fix them ... 556
The surface ... 556
Values and flow ... 557
Effects, capabilities and memory ... 557
Abstraction and concurrency ... 558

Appendix D — Grammar summary ... 558
Clause order of an op head ... 558
Declarations ... 559
Statements ... 559
Patterns ... 560
Expressions ... 560
Heads and closers ... 560

Appendix E — Standard library modules ... 560

`strings` — string view operations ... 563
Design and boundaries ... 563
Ops at a glance ... 563
Ops in detail ... 564
Using it ... 565
Counter-examples ... 566
Cautions ... 566

`strbuf` — owned string buffer and null-terminated `cstr` ... 568
Design and boundaries ... 568
Ops at a glance ... 569

Ops in detail 569
Using it 570
Counter-examples 570
Cautions 571

`fmt` — formatting assembled into the caller's buffer 572
Design and boundaries 572
Ops at a glance 572
Ops in detail 573
Using it 573
Counter-examples 574
Cautions 575

`utf8` — UTF-8 code point iteration and validation 576
Design and boundaries 576
Ops at a glance 577
Ops in detail 577
Using it 578
Counter-examples 578
Cautions 579

`utf16` — UTF-16 surrogate arithmetic 580
Design and boundaries 580
Ops at a glance 581
Ops in detail 581
Using it 582
Counter-examples 583
Cautions 583

`unicode` — Unicode property tables 584
Design and boundaries 584
Ops at a glance 585
Ops in detail 586
Using it 586
Counter-examples 587
Cautions 587

`codec` — hex and base64 589
Design and boundaries 589
Ops at a glance 590
Ops in detail 590
Using it 591
Counter-examples 591
Cautions 592

`regex` — Pike VM regular expressions without backtracking 593
Design and boundaries 593
Data structures 594
Ops at a glance 594
Ops in detail 595
Using it 595
Counter-examples 597
Cautions 597

`term` — terminal renderer (the pure half) 598

`sortlib` — in-place quicksort ... 601
`sortgen` — generic sorting (the type brings the comparison) ... 603
`searchlib` — binary search on sorted slices ... 605
`hashmap` — `u64` → `u64` hash map ... 607
`strmap` — string-keyed hash map ... 609
`vecs` — growable byte vectors with caller-held buffers ... 611
`spsc` — lock-free SPSC ring buffer ... 613
`hash` — hashing (hash map slots · corruption detection · SHA-256) ... 615
`math` — floating-point maths ... 616
`random` — random numbers (reproducible sequences · OS entropy) ... 617
`hmac` — HMAC-SHA256 and HKDF ... 619
`chacha` — ChaCha20 stream cipher ... 620
`poly` — Poly1305 one-time authenticator ... 621
`aead` — ChaCha20-Poly1305 seal and open ... 622
`x25519` — key agreement on a curve ... 623
`aes` — the AES-128 block cipher ... 625
`gcm` — AES-128-GCM authenticated encryption ... 626
`crypto_hw` — arithmetic the machine helps with ... 628
`bigint` — big-number modular arithmetic ... 630
`rsa` — RSASSA-PSS verification ... 631
`p256` — the NIST P-256 curve and ECDSA verification ... 632
`p384` — the NIST P-384 curve and ECDSA verification ... 633
`ecdsa` — ECDSA P-256 signing ... 635
`ed25519` — Ed25519 signature verification ... 636
`der` — minimal DER parser ... 637
`pem` — unwrapping PEM envelopes ... 638
`x509` — reading X.509 certificates ... 639
`verify` — certificate signatures and one link of a chain ... 641
- A chain is a line of links ... 641
- ops ... 642
- Design ... 642

`tls13` — the computational parts of TLS 1.3 ... 643
`tlssrv` — the TLS 1.3 server handshake ... 645
`tlscli` — the TLS 1.3 client handshake ... 647
- The order of the handshake ... 647
- ops ... 647
- Design ... 648

`http` — HTTP/1.1 request parser ... 650
`soa` — an SoA layout trial: one array per field ... 652

`allocs` — allocator traits, bumps and default allocators 654
`pool` — generational-handle block pool 657
`shard` — access units that split a store 660
`budget` — handle bit budgets and generation wraparound 662
`wire` — dividing one word into fields 664
`flags` — named on/off settings in one word 666
`segarena` — fixed-size segment arena 668
`pagecache` — page ids and pinning cursors 670
`growvec` — self-growing byte vector 672
`vecgen` — generic self-growing vector `vec t a` 674
`mapgen` — generic hash map `table k v` 676
`nodelist` — fixed intrusive lists 678
`segview` — cursors, total length and coalescing for segment views 680
`lifemode` · `lifeatom` — when a value ends 681
`io` — stream reading over slices 683
`outbuf` — buffered output where forgetting to flush is a compile error 685
`files` — file and directory streams where forgetting to close is a compile error 688
`tty` — terminal input 691
`net` — sockets (TCP loopback · in-process pairs) 694
`clock` — time and deadlines 696
`trust` — finding a trusted root in the trust store 698
It never holds the whole bundle 698
ops 698
Index 700

Preface

This book is a guide to Lowent, an experimental systems programming language. It rebuilds the manual that lived in the repository into a single book. Scattered manual chapters, explanations from the specification, standard library documents and proof commentary are laid out in one line of reading, and every example was actually run and checked.

The problem Lowent tries to solve fits in one sentence: **know what an op can and cannot do from its head alone.** Whether it is pure, what it receives and returns, what it does to the outside, and who permitted that are written in the head, and the compiler checks them. That goal explains nearly all of this language's unfamiliar surface — clauses closed with periods, capabilities handed over as arguments, heads written in only one order. So this book sets up the goal before the grammar, and tries to explain how each rule follows from it.

What this book promises and does not promise

Every example in this book is a real file in `docs/manual/examples/`. The top of each file says what must happen — whether it must give the same answer on the VM and natively, or which diagnostic must reject it — and the output left by the verification script after checking it is printed on the page as is. No run results were transcribed by hand.

Thanks to that discipline, several discrepancies between the compiler and the specification came to light while this book was written. Those that could be avoided by changing examples were avoided, and those readers might trip over are written in the text as defects. So this book describes **the language the repository's compiler actually accepts now**. Where it differs from the ideal the specification describes, it says so.

There are also things it does not promise. This language has not reached version 1, and its grammar and standard library still change. This book has its own edition number and follows the compiler as it changes. And this book has no exercises. It explains, asks questions and answers them. The ability to actually produce code grows only by writing it yourself. Fortunately this language's compiler rejects wrong code with coded diagnostics, so writing alone is less lonely than in other languages.

Who this book is for

It assumes readers who have programmed in at least one language. It does not explain what variables, loops and functions are; it explains **how they differ** in Lowent. If you know C, the parts on memory and the outside world are easy, and if you know Rust, the borrowing rules will be familiar. It is written to be readable without either. The proof commentary in Part X can be followed with high-school mathematics.

Editions and licence

The text of this book is released under CC BY-NC-SA 4.0 and the example code under MIT. The book comes out in Korean and English editions together, with the same chapter order and examples. The code and output of the examples are one set shared by both editions.

How to read this book

This book was written to be **read front to back**. Later chapters stand on the vocabulary and rules earlier chapters set up. The top of each chapter lists “what to know first” with chapter numbers, so if you skip ahead and get stuck, go back to that place.

But every reader’s situation differs. The paths below are **shortcuts**; pick the one closest to where you are.

★ Whichever path you choose, do not drop **Parts I and IV**. Part I sets up the goal and surface of this language, and Part IV the contracts, effects and capabilities that hold up that goal. Every other part stands on these two.

Path ① — learning this language from the start

The book’s order as is. Read Parts I to VIII in order, only the chapters you need from Part IX, and from Part X at least the last chapter (chapter 50).

Path ② — familiar with Rust or C and wanting only the differences

chapters 1–3 → chapters 4 and 11 → all of Part IV (chapters 14–17) → Part V (chapters 18–20) → chapters 22 and 23 → chapter 27 → chapter 29. For the rest of Parts II and III, skim only each chapter’s “Recap”, and read the chapter when a rule is new to you.

Path ③ — writing for machines without an operating system

Part I → chapter 4 → Part IV → chapters 18 and 20 → chapter 27 → chapters 29 and 30 → chapter 35. Where the heap is rejected (chapter 18) and machine tiers (chapter 30) are the heart of this path.

Path ④ — wanting to know how far this language’s claims are true

chapter 1 → chapter 14 → chapter 15 → chapter 12 → all of Part X. Each chapter of Part X states at its top which earlier chapters it connects to.

Devices in the text

Chapters open the same way — this chapter’s contents, what to know first, a question and answer that brings it out, why this chapter is here, what you gain at the end, and the questions

this chapter answers. The text contains questions and answers, common misconceptions, real cases, maths boxes and demonstrations (examples with actual output), and the recap at the end of each chapter gathers the points once more. Demonstration output is printed as the verification script left it. The long English diagnostics seen in the output are sentences the compiler actually produced.

Part I — Getting started

This part does not yet teach how to compute anything in Lowent. Instead it lays down **the place where you first meet this language**: why such a language is needed (chapter 1), how to build and run the tool and get rejected by it (chapter 2), and the rules of the surface that all code follows (chapter 3).

The goal (chapter 1) — sets the goal of knowing what an op can and cannot do from its head alone, and the five ideas that hold it up.

The tool (chapter 2) — runs the same program on the VM and natively and compares them, and shows contracts stopping execution and how to read diagnostic codes.

The surface (chapter 3) — the detached period, `do … end` blocks, prefix notation, and the clause order table for op heads. That table is used to the end of the book.

1 What Lowent sets out to do

In this chapter

1.1 Can you tell from the head alone? 20
1.2 Five ideas 22
1.3 One meaning, one spelling 23
1.4 Where the language stands 23
1.5 How to read this book 24

The need for this chapter, and its context

When the first chapter of a new language opens with syntax, the reader memorises words and misses **why those words look the way they do**. Most of Lowent's syntax follows from a single goal — **reading only the head of an op should tell you what that op can and cannot do.** With that goal in place first, the unfamiliar rules that come later (clauses closed by a full stop, `fn` marking purity, capabilities handed in as arguments) stop looking like arbitrary taste and start looking like faces of one design. So the goal and the way to read this book come first.

By the end of this chapter

You will learn what problem Lowent was built to solve and the five ideas that make up its answer (marking purity, contracts, effects paired with capabilities, memory without a garbage collector, and cross-checking two back ends). You will also settle that the language is still experimental, how the examples in this book are verified, and which devices to read it with.

The questions this chapter answers

1. If the head has to say that much, doesn't the code get longer?
2. If the grammar is that narrow, how do new features get in?

1.1 Can you tell from the head alone?

The thing people who fix programs do most is **find out what someone else's function does before calling it**. It is called `parse_header`, but does it open a file? Does it allocate? How does it report failure? Does it start a thread? In most languages the answers live in the **body**. You read the body, then the bodies of the functions it calls, then the level below that.

Lowent lifts those answers up into the **head**. Here is the first op head this book meets.

```
proc main input out cap io . output u8 . effects io .
```

Even without knowing the words, some things can be read. This op is a `proc` (it is not pure). It **receives** an `io` capability under the name `out`. It returns a `u8`. It performs an effect called `io`. What is not written can be read too — this op does not allocate (there is no `alloc`), has no file-system capability, and starts no threads (there is no `concurrent`). **What is not written cannot be done.** The compiler checks it.

Let us look at a small program that actually runs.

examples/ch01/heads.low

```
module heads .
rem run: main

rem fn: the same input always gives the same answer, and it touches neither the screen n
or files
fn area input w u64 . input h u64 . output u64 .
  requires le w 1000 .
  requires le h 1000 .
do
  return mul w h .
end

rem proc: may leave traces outside; what it does (effects io) and who allowed it (cap io
) are visible in the head
proc main input out cap io . output u8 . effects io .
do
  let a u64 be area 3 4 .
  let n u64 be write_out out 1 "area is twelve\n" .
  return narrow u8 a .
end
```

Output

```
$ lowentc --run main heads.low
area is twelve
main() = 12
```

- `module heads .` is this file's name tag. Every file starts with this one line.
- Lines starting with `rem` are words for people (comments). The compiler does not read them.
- `fn area … do … end` is an op that computes an area. `input w u64 .` means "take an unsigned 64-bit integer under the name `w`", and `output u64 .` is the type of the value returned. `requires le w 1000 .` is a promise: "`w` must be at most 1000". It is there so the product cannot overflow.
- `return mul w h .` returns `w` times `h`. The operation name comes first and the arguments follow. A statement ends with a detached period.
- `main` is the op where the program starts. `let a u64 be area 3 4 .` gets 12 into `a`, and `write_out out 1 "…"` writes a line to standard output (number 1). That line is possible because the head receives `cap io` and declares `effects io`. The 12 it returns becomes the exit value handed to the operating system.

Now let the head lie. An op declared as a pure `fn` tries to write to the screen on the quiet.

examples/ch01/mistake_hiddenio.low

```
module mistake_hiddenio .
rem expect: E-EFFECT-CALC

rem ✘ declared as fn, yet it tries to write to the screen on the quiet
fn area_loud input out cap io . input w u64 . input h u64 . output u64 .
  requires le w 1000 .
  requires le h 1000 .
do
```

```
  let n u64 be write_out out 1 "computing\n" .
  return mul w h .
end
```

Output

```
$ lowentc --check mistake_hiddenio.low
mistake_hiddenio.low:8:1 E-EFFECT-CALC: this fn is declared pure but performs `io` — mak
e it a `proc` with `effects …`, or remove the effect
```

The compiler refuses without running anything. The single word `fn` is a promise that "this op does not reach outside", and when the body breaks that promise, translation stops. So whoever calls a `fn` need not open its body. To fix it, say what is true: `proc area_loud … effects io ..`

> **Q. If the head has to say that much, doesn't the code get longer?**
>
> **A.** It does. The reading gets shorter in exchange. An op is written once, but it is read at every place that calls it, on every day someone fixes it. Lowent bets on that asymmetry. And because the compiler checks every clause of the head, the head does not go stale the way a comment does.

1.2 Five ideas

Five mechanisms make the head trustworthy. Parts II to VIII of this book unfold them one by one.

Idea	What the head says	Where
Separate pure from impure	`fn` or `proc`	chapter 2
Write promises and have them checked	`requires` · `ensures` · `errors`	chapter 14
Pair what is done with who allowed it	`effects` · `input …` `cap …`	chapters 15 and 16
Keep memory safe without a collector	`region` · `owned` · `ref`	chapters 18–20
Run two ways and compare	(a tool, not the head)	chapter 31

Table 1.1 — The five ideas behind Lowent

Marking purity. A `fn` is pure — the same inputs give the same output and it leaves no trace outside. A `proc` may not be. Which one it is is **always** written. If a pure `fn` tries to print, the compiler rejects it.

Contracts. An op writes what it requires of its caller (`requires`) and what it guarantees on return (`ensures`). When the values are constants, the check happens at translation time; otherwise it happens at run time, on entry and on return. A proven contract removes checks in the body — a contract is both documentation and grounds for optimisation.

Effects and capabilities. An op declares the marks it leaves on the world with `effects`, and is **handed** the capability to do that work as a `cap` input. Lowent has no ambient authority — no global output or global allocator that can be used secretly from anywhere. If a capability is not visible in the argument list, the op cannot do that work.

Memory without a collector. To use memory safely without a garbage collector, Lowent ties the lifetime of values to **regions** and checks a borrowing rule of one writer or many readers. For machines without an operating system there is a separate path that only allocates from a fixed window that never grows.

Cross-checking two back ends. The compiler, `lowentc`, runs the same intermediate representation two ways. One is a virtual machine (VM) that runs by itself; the other is native code, emitted as C and turned into machine code by a C compiler. If the two give different answers, that is a compiler defect. Every example in this book passed this cross-check.

> **A common misconception. Lowent proves everything statically**
>
> It does not. Borrowing, regions and integer widening are stopped at translation time, but bounds checks in unproven places, overflow, and contract checks are stopped **at run time**. It is a mixed design. What is proven and what is not is gathered in Part X — Chapter 50 is devoted entirely to **what is not proven**.

1.3 One meaning, one spelling

The **entropy** in Lowent's name is the measure of disorder. The language tries to reduce the places where one meaning can be written several ways. A few choices follow.

- A field is read only as `field p x`. There is no glued dot like `p.x`.
- Statements and clauses are closed with a free-standing full stop `.`, and a block is always `do … end`.
- The clauses of a head are written in one fixed order. A wrong order is rejected, and `--fmt` fixes it.
- Removed words (`loop`, `give`, `on` and so on) are not quietly accepted; they are rejected with `E-VOCAB-REMOVED`.

With one spelling per meaning, people and tools read the same code in the same shape. The examples in this book all look alike not because of the author's taste but because the language only allows that shape.

> **Q. If the grammar is that narrow, how do new features get in?**
>
> **A.** Instead of adding words, a feature comes in **as one more clause in a frame that already exists**. `pipe` is not a new loop syntax but a frame that writes one word per line inside a `do … end` block (chapter 24), and parallelism is not a new construct but a `parallel` clause in an op head (chapter 27). That is why the clause-order table is one of the most important tables in this language (chapter 3).

1.4 Where the language stands

Lowent is an **experimental language**. It has not reached version 1, and its grammar and standard library still change. This book is written against the grammar the compiler in the repository actually accepts today, and when the compiler changes the book follows. That is why the book has its own edition number.

The compiler is written in C23, and its only external dependency is a vendored copy of `proven_c_lib`. Native code is emitted as C and built with the system's C compiler, so it usually runs wherever there is a C compiler, and microcontrollers without an operating system are among its targets (chapter 30).

> **In practice. How the examples in this book are verified**
>
> Every `.low` file in this book lives in `docs/manual/examples/`, and each carries a one-line directive at the top. With `rem run: …` it must be run on both the VM and native code and give the same output; with `rem expect: E-…` the compiler must reject it with exactly that diagnostic; with no directive it must pass `--check`. The run results on the page were not copied by hand — they are exactly what the verification script left behind.

1.5 How to read this book

This book was written with two kinds of reader in mind.

Readers new to programming read from Part I in order. Every piece of syntax comes with both "what it does" and "why it looks this way", and each example is followed by an explanation of what its lines do. The "Common mistakes" at the end of Chapters 2 to 37 show the errors beginners really run into, with their diagnostics as they are. There is no need to fear error messages. In this language a diagnostic does not say "you are wrong"; it says "this is where the code differs from its promise". Learn the first words from the table below.

Readers who have used other languages can follow just the example code and each chapter's "syntax at a glance" table. The examples and that table show the chapter's syntax together. If you know C, Parts V and VIII will come easily; if you know Rust, the borrowing rules of Part V will look familiar. Places where an idea carried over from a familiar language goes wrong here are collected in the "A common misconception" boxes.

Word	Meaning
source · module	one `.low` file a person writes · the named unit that file forms (chapter 21)
compiler · translation	the program that reads the source, checks it and turns it into something runnable (`lowentc`) · that work
diagnostic	an error (`E-`), warning (`W-`) or note (`N-`) from the compiler. Its name is the key to look it up (Appendix B)
value · type	one piece of data such as a number or text · what kind it is and how many bits (`u64` is an unsigned 64-bit integer)
local name	a name given to a value: `let` if it never changes, `var` if it does (chapter 6)
op	a named piece of work; a function in other languages. `fn` if pure, `proc` otherwise (chapter 5)
effect · capability	the trace an op leaves outside (`effects io`) · the permission to do that (`cap io`) (chapters 15 and 16)
contract	conditions an op requires before it takes a call and ensures when it returns (`requires` · `ensures`) (chapter 14)

Table 1.2 — First words

Every chapter opens the same way. It lists the earlier chapters it leans on (**What to know first**), asks one question that makes you recall them, says why the chapter sits where it does and what you will have by the end, and then shows the questions the chapter answers. The devices you meet inside the text are these.

Device	Use
Q & A	A question likely to come up while reading, and its answer
A common misconception	A plausible but wrong idea, and why it is wrong
In practice	Something that actually happened in the language or its standard library
The mathematics	A formal explanation you can skip without losing the thread
Demonstration	An example file and the actual output left by the verification script

Table 1.3 — Devices used in the text

On a first reading, take Parts I to IV in order. After that you can jump to whichever part you need. For the standard library go to Part IX; to find out how far this language's claims are true, go to Part X.

2 A first program — build, run, get rejected

In this chapter

2.1 Building the tool 26
2.2 hello, entropy 27
2.3 Running it two ways 28
2.4 How a contract stops execution 28
2.5 Getting rejected 29
2.6 `--fmt` fixes the shape 30
2.7 Arguments and capabilities 31
2.8 Common mistakes 31
2.9 This chapter's syntax at a glance 34

What to know first

chapter 1, What Lowent sets out to do · knowing what an op can do from its head alone

Looking back

What did chapter 1 read from the head `proc main input out cap io . output u8 . effects io .`? And what did it read from **what was not written**?

> **A.** That the op is impure (`proc`), receives an `io` capability under the name `out`, returns a `u8`, and performs the `io` effect. From what was not written: the op does not allocate, has no file-system capability and starts no threads. This chapter actually builds and runs a program with that head.

The need for this chapter, and its context

The first step in learning a language is knowing **what the tool in your hand does for you**. Lowent's tool, `lowentc`, has no default mode, runs the same program two ways (VM and native), and answers wrong programs with diagnostics that carry codes. With these three in your fingers before the grammar, you can run and get rejected by every new rule in later chapters yourself. That is why this chapter comes before the description of the surface syntax (chapter 3).

By the end of this chapter

You will build `lowentc` and run the smallest program once on the VM and once natively. You will see why the returned value becomes the exit code, how a contract stops execution, and how to run `test` blocks. You will also learn how to read the diagnostics the compiler produces when it rejects a program, and how far `--fmt` will fix the shape for you.

The questions this chapter answers

1. Why is there no default mode? Couldn't `lowentc hello.low` just build it?
2. Do I have to memorise diagnostic codes?

2.1 Building the tool

The compiler is called `lowentc`. After getting the repository, build it in `impl/`. All you need is a C compiler that knows C23 (gcc or clang) and `make`.

```
cd impl
make            # build/lowentc
make check      # unit tests and standard-library checks
```

When `make` finishes there is an `impl/build/lowentc`. Commands in this book are written as `lowentc`, assuming it is on your path.

Mode	What it does
`--check`	Checks only (contracts, effects, ownership, capabilities). Emits nothing
`--run OP args…`	Runs one op on the VM. Slice arguments are given as `[3,4,8]`
`--emit-c`	Writes native C source to standard output
`--test`	Runs `test` blocks

`--fmt`	Prints the canonical form
`--diag-json`	Writes diagnostics as one JSON object per line (for tools and AI)

Table 2.1 — Common `lowentc` modes

> **Q. Why is there no default mode? Couldn't `lowentc hello.low` just build it?**
>
> **A.** It could, but then **what the command did** would not be visible on the command line. Whether it only checked, ran, or wrote files would have to be remembered. This is chapter 1's "one meaning, one spelling" applied to the tool. If you want something shorter, subcommand names such as `lowentc check` and `lowentc run` do the same work.

2.2 hello, entropy

The smallest program that prints.

examples/ch01/hello.low

```
module hello .
rem run: main

proc main input out cap io . output u8 . effects io . do
  return narrow u8 (write_out out 1 "hello, entropy!\n") .
end
```

Output

```
$ lowentc --run main hello.low
hello, entropy!
main() = 16
```

The first line, `module hello .`, is the module declaration every file opens with. The `rem` on the second line is a line comment; in this book it tells the verification script what to do.

Read the head one clause at a time.

- `proc main` — an impure op named `main`. Output leaves a trace outside, so it cannot be a `fn`.
- `input out cap io .` — receives an `io` capability named `out`. It is needed to write to standard output.
- `output u8 .` — returns a `u8`.
- `effects io .` — declares that it performs input/output.

The body lies between `do` and `end`. `write_out out 1 "…"` takes the capability `out` as its first argument, writes bytes to file descriptor `1` (standard output), and returns **the number of bytes written**. `hello, entropy!\n` is 16 bytes, so it returns 16. That is narrowed to fit a `u8` with `narrow u8`. Narrowing carries a check that stops execution if the value does not fit (chapter 4).

The last line of the output, `main() = 16`, is the line where the VM shows the op's return value.

2.3 Running it two ways

`--run` ran it on the virtual machine inside the compiler. To run the same program natively, emit C and build it with a C compiler.

```
lowentc --emit-c hello.low > hello.c
cc -O2 -o hello hello.c -lm
./hello main
```

The native executable takes the name of the op to run as its first argument. Running `main` prints the same line, and the return value 16 comes out as **the process exit code** (see it with `echo $?`). That is why `main`'s `output` is `u8` — an exit code is one byte.

> **In practice. When the two results differ, whose fault is it?**
>
> The VM and native code start from the same intermediate representation. If they produce different output, it is a defect in the **compiler**, not the program. The development repository runs hundreds of ops on both back ends with boundary-value arguments and compares them byte for byte, and this book's verification script also runs every example carrying `rem run:` twice and compares. One discrepancy surfaced while writing this book — `arg`, which reads program arguments, counts positions differently on the VM and in native code. So examples that read arguments are only checked in this edition, and their run results are not shown.

Slice arguments are given in brackets. The next op computes the mean of a byte slice.

examples/ch01/stats.low

```
module stats .
rem run: mean [3,4,8]

fn mean input xs slice u8 . output u64 .
  requires gt (len xs) 0 .
do
  var total u64 be 0 .
  for x xs do
    set total (add total (widen u64 x)) .
  end
  return div total (len xs) .
end
```

Output

```
$ lowentc --run mean stats.low [3,4,8]
mean([3,4,8]) = 5
  arg0 (written) = [3,4,8]
```

`requires gt (len xs) 0 .` is a **contract**. The caller promises not to pass an empty slice. Because of that promise, the body's `div total (len xs)` need not worry about dividing by zero. The line `arg0 (written) = [3,4,8]` below the result is the VM showing the final state of the slice argument.

2.4 How a contract stops execution

What happens if a contract is broken? The next op only accepts values up to 100.

examples/ch02/twice.low

```
module doubling .
rem run: twice 21
rem test

export fn twice input n u32 . output u32 .
  requires le n 100 .
do
  return mul n 2 .
end

test twice_works
do
  expect eq (twice 5) 10 .
  expect eq (twice 0) 0 .
end
```

Output

```
$ lowentc --run twice twice.low 21
twice(21) = 42
$ lowentc --test twice.low
  [PASS] twice_works
== tests: 1 run, 1 passed, 0 FAILED ==
```

Give it a value that breaks the promise, as in `lowentc --run twice twice.low 200`, and the VM stops before entering the body with `E-VM-CONTRACT: requires violated at entry`. If the argument is written as a constant in a call (`twice 250`), there is no need to run at all — `--check` rejects it with `E-CONTRACT-IMPOSSIBLE`. A contract is not a comment.

The `test twice_works` block in the same file is a test. `expect` is the assertion, and `--test` runs every `test` block. The last two lines of output are its result. `--check` does not run tests — passing the check does not mean the tests passed, so running `--check` on a file with tests produces a `W-TEST-NOT-RUN` warning to say so.

> **A common misconception. If `--check` is quiet, the program is correct**
>
> What `--check` guarantees stops at **the promises written in the head and the body not contradicting each other**. Promises that were not written cannot be checked, tests were not run, and contract violations that only show at run time only show when you run. “The check passed”, “the tests passed” and “it is correct” are three different statements.

2.5 Getting rejected

A good part of the time spent learning Lowent is time spent being rejected by the compiler. Rejection is not punishment but conversation. Every diagnostic carries a stable code, and that code keeps its meaning across releases.

examples/ch01/twice_bad.low

```
module twice_bad .
rem expect: E-EFFECT-CALC
```

```
fn shout input out cap io . output u64 .
do
  return write_out out 1 "hi\n" .
end
```

Output

```
$ lowentc --check twice_bad.low
twice_bad.low:5:1 E-EFFECT-CALC: this fn is declared pure but performs `io` — make it a
`proc` with `effects …`, or remove the effect
```

A diagnostic line has the shape `file:line:column code: explanation`. In this example `fn shout` is declared pure but its body does input/output with `write_out`. The compiler rejects it with `E-EFFECT-CALC` and names two ways to fix it — make it a `proc` with `effects`, or remove the effect.

With `--diag-json` the same diagnostic comes out as one line of JSON. The rule code (`rule`), location (`span`) and the name of the repair (`repair`) are separate fields, so an editor or an AI can tell what is wrong without picking the prose apart.

```
{"rule":"E-EFFECT-CALC","sev":"error","phase":"effect", … ,"repair":"R-CALC-TO-PROC"}
```

> **Q. Do I have to memorise diagnostic codes?**
>
> **A.** No. Codes are names to search for and refer to. The appendices of this book collect the diagnostics you will meet often, and each chapter shows in the text the codes that come out when its rules are broken.

2.6 `--fmt` fixes the shape

The clauses of an op head are written in one fixed order. Inputs come before the output. A wrong order is rejected.

examples/ch02/messy.low

```
module messy .
rem expect: E-CLAUSE-ORDER

fn area output u64 . input w u64 . input h u64 .
do
  return mul w h .
end
```

Output

```
$ lowentc --check messy.low
messy.low:4:22 E-CLAUSE-ORDER: a data input comes after `output`. An op header has ONE o
rder: `satisfies`/`lowdoc` · `vector`/`priority` · `comptime` inputs · capability/region
 inputs · `using` · data inputs · `output` · `effects` · `link`/`variadic` · `asm` · `ac
cess`/`inplace`/`invalidates`/`parallel`/`reduce` · `requires` · `ensures` · `errors` ·
`tests` (`--fmt` moves the non-input clauses for you; inputs are call positions, so reor
der those and their call sites yourself)
```

The long explanation in the diagnostic is the order table itself. The whole of it is covered in chapter 3 for now remember one thing: `--fmt` moves **clauses that are not inputs** (output, effects, contracts and so on) into place. It does not reorder inputs among themselves. The order of inputs is also the order of arguments at every call site, so moving them would change every call.

The canonical form `--fmt` prints puts one clause per line and shows every parenthesis. Nobody has to write in that shape. The canonical form is the yardstick a machine uses to compare **whether two pieces of code mean the same**.

2.7 Arguments and capabilities

Reading program arguments is a capability too. The next program greets its first argument.

examples/ch02/greet.low

```
module greet .
rem (argument numbering differs between native and VM, so this is only checked)

proc main input out cap io . input a cap args . output u8 . effects io .
do
  let who option slice u8 be arg a 0 .
  guard is_some who . else return narrow u8 (write_out out 1 "no name\n") .
  let n u64 be write_out out 1 "hello, " .
  let m u64 be write_out out 1 (some_value who) .
  let k u64 be write_out out 1 "\n" .
  return 0 .
end
```

Output

```
$ lowentc --check greet.low
== check: ok ==
```

`input a cap args .` receives the argument capability, and `arg a 0` reads the first argument. There may be no argument, so the result is `option slice u8`. `guard is_some who . else …` leaves on the spot when there is no value, and below it you may trust that the value exists (chapter 11). Capability inputs come before data inputs, and the two capabilities take argument positions in the order written.

Note that the single line `effects io .` covers both capabilities. `args` is a capability that can be read without an effect. How capabilities pair with effects is tabulated in chapter 16.

2.8 Common mistakes

These are the rejections and stops you meet most often in a first program. When you see one of these codes, come back here.

> **Counter-example. Opening `do` and never closing it with `end`**
>
> **examples/ch02/mistake_noend.low**

```
module mistake_noend .
rem expect: E-BLOCK-UNCLOSED

proc main input out cap io . output u8 . effects io .
do
  let n u64 be write_out out 1 "hi\n" .
  return narrow u8 n .
rem ✘ the `do` that opened the body has no matching `end`
```

Output

```
$ lowentc --check mistake_noend.low
5:1 E-BLOCK-UNCLOSED: missing 'end' for do-block
```

A body opens with `do` and closes with `end`. If the file ends before the closer, the compiler cannot tell where the body stops and reports `E-BLOCK-UNCLOSED`. The `5:1` in the diagnostic is the position of the `do` left without a partner — when bodies are nested, match the pairs starting from that line. Consistent indentation makes the pairs easy to see.

Counter-example. Printing without receiving the capability

examples/ch02/mistake_nocap.low

```
module mistake_nocap .
rem expect: E-EFFECT-NO-CAP

rem ✘ declares `io` but never receives the right to do it (`cap io`)
proc main output u8 . effects io .
do
  let n u64 be write_out 1 "hi\n" .
  return narrow u8 n .
end
```

Output

```
$ lowentc --check mistake_nocap.low
mistake_nocap.low:5:0 E-EFFECT-NO-CAP: this op declares the `io` effect but receives NO
capability that authorizes it. `io` is a cap-effect (RFC-0007 §6.7): I/O is a RIGHT you
are HANDED, not an ambient power — `input fs cap file_system .` (or another `cap …` inpu
t). An effect you declare but hold no capability for is a claim the signature cannot bac
k
```

`effects io .` **declares** that the op does I/O; what actually **allows** it is the capability received as an input (`input out cap io .`). Declaring the effect without the capability is `E-EFFECT-NO-CAP`. I/O is not a power you can grab anywhere: it is a right handed to `main` when the program starts and passed on to the ops that need it (chapter 16). The fix: add `input out cap io .` to the head and pass `out` as the first argument of `write_out`.

Counter-example. Returning an exit code larger than 255

examples/ch02/mistake_exitcode.low

```
module mistake_exitcode .
rem trap: main

proc main input out cap io . output u8 . effects io .
do
  var n u64 be 0 .
  var i u64 be 0 .
  while lt i 8 . do
    set n (add n (write_out out 1 "32 bytes of text on every line.\n")) .
    set i (add i 1) .
  end
  rem ✘ 256 bytes were written --- an exit code is one byte (0 … 255), so it does not fi
t
  return narrow u8 n .
end
```

Output

```
$ lowentc --run main mistake_exitcode.low
32 bytes of text on every line.
32 bytes of text on every line.
32 bytes of text on every line.
32 bytes of text on every line.
32 bytes of text on every line.
32 bytes of text on every line.
32 bytes of text on every line.
32 bytes of text on every line.
== ir diagnostics (1) ==
0:0 E-VM-CAST: value does not fit the target width (use narrow_wrap / narrow_sat / narro
w_try)
```

This compiles, but stops while running. The value `main` returns is the **exit code** handed to the operating system, which is one byte (`u8`), and 256 does not fit in `narrow u8`. Lowent does not quietly truncate it to 0; it stops right there, because a truncated exit code reads as “success” and hides the bug. If what you want to know is the number of bytes written, print it, and use the exit code only for success (0) or failure (non-zero).

Counter-example. Misspelling the op to run

examples/ch02/mistake_opname.low

```
module mistake_opname .
rem trap: mian

proc main input out cap io . output u8 . effects io .
do
  let n u64 be write_out out 1 "hi\n" .
  return narrow u8 n .
end
```

Output

```
$ lowentc --run mian mistake_opname.low
== ir diagnostics (1) ==
0:0 E-VM-UNDEF: no such op
```

The name after `--run` must match an op in the file exactly. There is no `mian`, so the VM stops with `E-VM-UNDEF: no such op`. A native executable also exits non-zero when the op named by its first argument does not exist. This is the price of keeping what runs visible on the command line — in return, you can run any op in the file on its own.

2.9 This chapter's syntax at a glance

Shape	Meaning	Why
`module hello .`	the module name of this file	the name other files use in `use hello .`
`rem …`	line comment (to the end of the line)	a word rather than a symbol — this book also uses it for check directives
`proc main input out cap io . output u8 . effects io .`	where the program starts	rights received, exit code returned and what it does are all in the head
`do … end`	a body	an opener and a closer that pair up
`write_out out 1 "…"`	write to standard output (1) and return the byte count	the capability `out` comes first — no right, no writing
`narrow u8 n`	narrow to `u8` (stops if it does not fit)	so values never change silently
`requires c . • test t do … end • expect c .`	contract · test block · assertion inside a test	promises are checked; tests run separately
`lowentc --check f.low`	check only	the command line shows what was done — there is no default mode
`lowentc --run op f.low args…`	run one op on the VM	try any op in the file on its own
`lowentc --emit-c f.low > f.c`	emit C for a native build	so the result can be compared with the VM
`lowentc --hw auto f.low`	carry the machine's crypto instructions and its width (AES-NI, carry-less multiply, SSE2/AVX2 lanes)	the build picks what to CARRY; when it carries more than one, the program chooses once at start — the answer is the same either way

`lowentc --test · --fmt`	run tests · reprint in canonical form	checking, testing and layout are different jobs

Table 2.2 — The shape of a first program and the tool — shape · meaning · why it looks this way

Recap

`lowentc` always takes an explicit mode. `--run` runs the program on the VM, `--emit-c` plus a C compiler runs it natively, and the two must agree. `main`'s return value is the exit code. Contracts stop execution before running or on entry, and `test` blocks run only under `--test`. Rejections carry stable codes, and `--fmt` fixes clause order only for clauses that are not inputs.

3 The surface — full stops, blocks and clause order

In this chapter

3.1 The name first, then the arguments 36
3.2 A free-standing full stop closes 36
3.3 A block is always `do … end` 38
3.4 Comments and literals 39
3.5 Names 42
3.6 Words are a budget 42
3.7 The boundaries of the `expr` island 43
3.8 Clause order in an op head 43
3.9 Common mistakes 45
3.10 This chapter's syntax at a glance 48

What to know first

chapter 1, What Lowent sets out to do · one meaning, one spelling
chapter 2, A first program · `--check` diagnostics and `--fmt`

Looking back

In chapter 2, what rejected `fn area output u64 . input w u64 . …`, and how far does `--fmt` fix it?

A. `E-CLAUSE-ORDER` rejected it, because a data input came after `output`. `--fmt` moves clauses that are not inputs (output, effects, contracts) into place but leaves the order of inputs alone, because the order of inputs is also the order of the caller's arguments. This chapter covers that whole order table and the smaller rules underneath it — full stops, parentheses, blocks.

The need for this chapter, and its context

Lowent code looks strange at first. There are more full stops than parentheses, and words come before operands instead of operators between them. Memorising that strangeness as a list of exceptions takes a long time. But the rules of the surface are few, and all of them follow from chapter 1's "one meaning, one spelling". Every later chapter is written on this surface, so it is gathered here in one place before values and flow (Part II).

By the end of this chapter

You will learn that a free-standing full stop closes a form while a newline closes nothing, that notation is prefix with the name first, and where the boundaries of the `expr` island that allows infix lie. You will pick up the rules for comments and literals, that names contain no dots and shadowing is forbidden, and that every block is `do … end`. Finally you will understand the whole clause-order table of an op head and why it is ordered as it is.

The questions this chapter answers

1. When arithmetic gets long, don't the parentheses pile up and become hard to read?
2. Can a module and an op share a name?

3.1 The name first, then the arguments

Lowent expressions use **prefix notation**. The name of the operation comes first and its arguments follow. `add a b` adds `a` and `b`, and a form inside a form is wrapped in parentheses.

```
let total u64 be add 1 2 .
let mixed u64 be add 1 (mul 2 3) .
```

Prefix notation has no precedence. Someone reading `1 + 2 * 3` knows that multiplication comes first because they **memorised** it, but in `add 1 (mul 2 3)` the parentheses already say so. The principle is the same for ops you write yourself. `write_out out 1 "hi"`, `field p x` and `mean xs` all put the name first.

Q. When arithmetic gets long, don't the parentheses pile up and become hard to read?

A. That is why there is an `expr` island. Inside a place starting with `expr`, the four arithmetic operators, comparisons, `and` and `or` may be written infix. An expression in the island is translated to **exactly the same meaning** as the prefix form and costs nothing at run time. The island's boundaries appear later in this chapter and in detail in chapter 8.

3.2 A free-standing full stop closes

The end of a statement or clause is closed by a **free-standing** full stop `.`, called a **closer**. A newline is not a closer — it is whitespace, exactly like a space. So wherever you break a line, the meaning stays the same.

examples/ch03/poly.low

```
module newline .
rem run: poly 5 4
rem run: poly_expr 5 4

note WHY
  개행은 공백이다. 폼은 자기 닫개인 떨어진 마침표에서 끝난다.
  그래서 줄을 어디서 나누어도 뜻이 같다.
WHY

fn poly input a u64 . input b u64 . output u64 .
do
  return add (mul a 2)
             (mul b 3) .
end

fn poly_expr input a u64 . input b u64 . output u64 .
do
  return expr a * 2 + b * 3 .
end
```

Output

```
$ lowentc --run poly poly.low 5 4
poly(5, 4) = 22
$ lowentc --run poly_expr poly.low 5 4
poly_expr(5, 4) = 22
```

The `return` in `poly` spans two lines but is one form; it ends at the full stop at the end of the second line. That is why there is no line-continuation marker (C's trailing backslash, a trailing comma, indentation rules). `poly_expr` in the same file writes the same computation as an `expr` island, and the two ops give the same answer.

`note WHY … WHY` is a multi-line comment; it ends at a line where the word written after `note` stands alone. `rem` is a line comment. The language has no symbolic comments like `//` or `/* */` — comments too are opened with words.

The number of full stops works as a kind of **checksum**. If opened forms and closers do not match, the compiler says so. Forgetting one parenthesis looks like this.

examples/ch03/dots.low

```
module dots .
rem expect: E-GROUP-UNCLOSED

fn poly input a u64 . input b u64 . output u64 .
do
  return add (mul a 2 (mul b 3) . .
end
```

Output

```
$ lowentc --check dots.low
7:1 E-PAREN-ESCAPE: a `(` opened inside this block was never closed — it would have ESCA
PED past the `end`. It used to: the newline-continuation stayed on for the REST OF THE F
ILE and every op after it was swallowed into the group. A bracket may not cross a block
```

```
boundary; it is closed here
6:14 E-GROUP-UNCLOSED: missing ')'
```

The first diagnostic means a parenthesis tried to leak past the block's `end`. Parentheses cannot cross a block boundary. So a single wrongly closed parenthesis stops inside that block instead of silently swallowing the rest of the file.

> **A common misconception. A semicolon closes statements too**
>
> It once did. When the processor met `;` it literally produced a full stop. Two spellings of one meaning force a reader to know both, so it was removed. Today `;` is rejected with `E-VOCAB-REMOVED`.

examples/ch03/semi.low

```
module semi .
rem expect: E-VOCAB-REMOVED

fn twice input a u64 . output u64 . do return mul a 2 . ; end
```

Output

```
$ lowentc --check semi.low
4:57 E-VOCAB-REMOVED: `;` was a THIRD spelling of the closer `.` (the lexer literally em
itted a DOT for it) — one meaning, three spellings, and SPEC-002 §2.5 forbids synonyms.
It only survived inside the old interpreter's second language. Write `.`
```

3.3 A block is always `do` … `end`

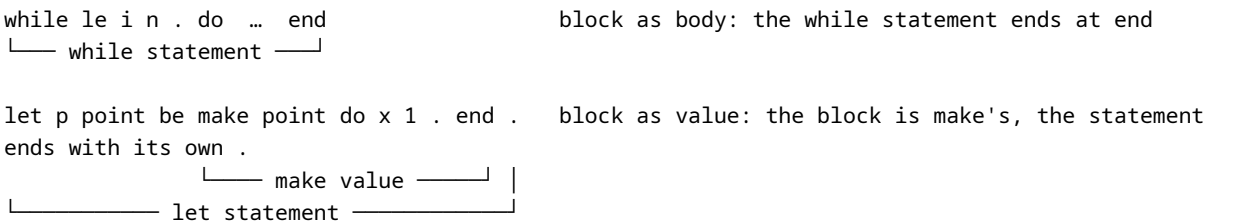
Every place that groups statements opens with `do` and closes with `end`: an op body, the body of `if`, `while` and `for`, the arms of `match`, and the bodies of declarations such as `struct`, `enum`, `trait` and `actor`.

```
struct point do
  x u64 .
  y u64 .
end

fn sum_to input n u64 . output u64 .
do
  var total u64 be 0 .
  var i u64 be 1 .
  while le i n . do
    set total (add total i) .
    set i (add i 1) .
  end
  return total .
end
```

`do … end` works like a pair of braces. `end` closes **only its own `do`** and leaves everything outside alone. So there are just two rules to know.

- **A construct that owns a block as its body** ends at `end`: op declarations, `struct`·`enum`, `if`·`while`·`for`·`match`. As in the example above, no full stop follows the `end` — one there closes nothing and is rejected with `E-DOT-STRAY`.
- **A statement that uses a block as a value** ends with its own full stop, like any statement. Building a struct value with `make` and binding it with `let` is the usual case: `let p point be make point do x 1 . y 2 . end .` — the last stop belongs to the `let`. Leaving it out is `E-DOT-MISSING`.

In C terms: no `;` after `if (c) { … }`, but one after `p = (struct point){ 1, 2 };`.



Each statement inside a block must end with its own full stop too; `end` does not close an open statement for you. A bare `do … end` with nothing opening it is not allowed either (`E-BLOCK-NOHEAD`) — a block always has a head. Opening a declaration block with only a line break instead of `do` is rejected with `E-STMT-NODO`, and `--fmt` inserts the `do`. The clauses of an op header work the same way. Each ends with its own stop — `input n u64 .`·`output u64 .` — and the next clause word does not close the one before it; leaving the stop out is `E-DOT-MISSING`.

3.4 Comments and literals

Kind	Examples
Integers (decimal, hex, binary)	`42` · `0xFF` · `0b1010` · `1_000_000`
Floating point	`3.14` · `6e-3` · `0x1.8p3`
Strings (bytes)	`"hello\n"`
Strings (UTF-16 · code points)	`u"…"` · `U"…"`
Booleans · absence	`true` · `false` · `none`

Table 3.1 — Writing literals

An integer literal has no type of its own; the place where it is used gives it one. A leading 0 is not octal (`0755` is 755). A literal that does not fit the type of its place is rejected rather than silently truncated.

examples/ch03/lit300.low

```
module lit300 .
rem expect: E-TYPE-WIDTH

fn f output u8 .
do
  let x u8 be 300 .
  return x .
end
```

Output

```
$ lowentc --check lit300.low
lit300.low:6:0 E-TYPE-WIDTH: value does not fit the declared type (a literal out of rang
e, or a narrowing — use narrow / narrow_wrap / narrow_sat / round_to)
```

C truncates 300 to 44 when it is put in a `u8`. Then the value written in the source and the actual value differ. Lowent stops it at translation time. Shrinking is done by **writing** one of the named `narrow` operations (chapter 4).

The set of string escapes is closed. There are the familiar `\n`, `\t`, `\\` and `\"`, plus `\xNN` (exactly two digits), `\uXXXX` (four) and `\UXXXXXXXX` (eight); any other escape is rejected with `E-STR-ESCAPE`. C's `\x` eats hex digits without end, so the meaning of `"\x41e"` depends on the character next to it; in Lowent `\x41e` is always `A` followed by `e`. A string is not a separate type but a byte slice (`slice u8`).

Written	Meaning	Written	Meaning
`\\`	backslash	`\v`	vertical tab
`\"`	double quote	`\0`	zero byte
`\'`	single quote	`\xNN`	two hex digits — any single byte
`\a`	alert	`\uXXXX`	code point, four digits
`\b`	backspace	`\UXXXXXXXX`	code point, eight digits
`\f`	form feed	`\n`	newline
`\r`	carriage return	`\t`	horizontal tab

Table 3.2 — String escapes — these fourteen only

`\uXXXX` and `\UXXXXXXXX` write a code point, and the prefix decides how many units carry it — `"\U0001F600"` is 4 bytes, `u"…"` is 2 UTF-16 code units (a surrogate pair) and `U"…"` is 1 code point. Surrogate values (D800 … DFFF) and values above 10FFFF are rejected. There are no octal escapes — the language has no octal notation at all, so reviving it only in literals would make that the one exception.

Using every shape in the table in one file looks like this.

examples/ch03/literals.low

```
module literals .
rem run: ints
rem run: floats
rem run: chars
rem run: texts

note END
note 와 같은 태그 사이의 줄은 모두 주석이다.
Lines between note and the same tag are all a comment.
END

rem integers: binary, digit separator _, hexadecimal; a leading 0 is not octal
fn ints output u64 . do
  let a u64 be 0b1010 .
  let b u64 be 1_000_000 .
  let c u64 be 0xFF_FF .
  let d u64 be 0755 .
  return add (add a b) (add c d) .
end
```

```
rem floating point: exponent e, hexadecimal float p (0x1.8 is 1.5, p1 multiplies by 2)
fn floats output f64 . do
  let e f64 be 1.5e3 .
  let h f64 be 0x1.8p1 .
  return add e h .
end

rem characters: 'A' is one byte, u'가' one UTF-16 code unit, U'😀' one code point
fn chars output u64 . do
  let a u8 be 'A' .
  let k u16 be u'가' .
  let s u32 be U'😀' .
  return add (widen u64 a) (add (widen u64 k) (widen u64 s)) .
end

rem strings: \x41 takes exactly two digits so "\x41e" is two bytes; u"가나" is two code
units
fn texts output u64 . do
  let x u64 be len "\x41e" .
  let w u64 be len u"가나" .
  rem text TAG … TAG holds several lines as written; a \n in the body is not unescaped
  let h u64 be len text DOC
line one
a\nb
DOC .
  return add (mul x 100) (add (mul w 10) h) .
end
```

Output

```
$ lowentc --run ints literals.low
ints() = 1066300
$ lowentc --run floats literals.low
floats() = 1503.0
$ lowentc --run chars literals.low
chars() = 172609
$ lowentc --run texts literals.low
texts() = 233
```

- `ints` is 10 (`0b1010`) + 1000000 (`1_000_000`) + 65535 (`0xFF_FF`) + 755 (`0755`). An underscore goes only **between** digits and does not change the value; it makes long numbers easier to read. `0755` is 755, not 493, a choice that removes a C trap.
- `floats` is 1500 (`1.5e3`) + 3 (`0x1.8p1` — 1.5×2^1). Hexadecimal floats are for when the bits must be written exactly. Decimal `0.1` cannot be written exactly in binary, but `0x1.8p1` is exactly 3.
- `chars` is 65 (`'A'`) + 44032 (`u'가'`) + 128512 (`U'😀'`). The prefix decides the width of the element. Writing a character that does not fit in one byte, such as `'가'`, without a prefix is refused with `E-CHAR-WIDTH`.
- In `texts`, `"\x41e"` is 2 bytes, `u"가나"` is 2 code units and `text DOC … DOC` is 13 bytes. A multi-line string (heredoc) holds, **exactly as written**, everything from the tag after `text` to the line where the same tag stands alone. The `\n` in the body is two characters, not a newline, and no newline is added after the last line. It is there to paste long descriptions or test input without escapes.

- The `note END … END` at the top is a multi-line comment. It turns a whole block into a comment without `rem` on every line.

3.5 Names

A name starts with an ASCII letter or underscore and continues with letters, digits and underscores. Two things differ from other languages.

First, **declared names contain no dots.** A place with a dot attached is a path **referring** to a name, and it has only three meanings — a module's name (`allocs.byte_allocator`), a variant's name (`node.lit`), and the name of a declaration attached to a type (`rect.area`). Looking inside a value (reading a field, calling a method) is not a glued dot but a form: `field p x · method s area`.

Second, **there is no shadowing.** An inner name that reuses the spelling of an outer name is rejected.

examples/ch03/shadow.low

```
module shadow .
rem expect: E-NAME-SHADOW

fn f input n u64 . output u64 .
do
  let n u64 be add n 1 .
  return n .
end
```

Output

```
$ lowentc --check shadow.low
shadow.low:6:0 E-NAME-SHADOW: a local binding takes the name of a PARAMETER of the same
op. The namespace is FLAT (SPEC-002 2.10: no shadowing) — from here on the same letters
mean the new binding, and a reader who saw the signature will read the wrong value. Rena
me the local
```

A shadowed name makes someone who read the head **think of a different value**. If you had to work out on every line whether `n` is the parameter or the new local, that is exactly semantic entropy. No parameter, module name, builtin op name or name still alive in an outer block can be shadowed. Choose a new name instead.

> **Q. Can a module and an op share a name?**
>
> **A.** No. With an op `twice` inside module `twice`, both would be referred to as `twice.twice` and `twice` and could not be told apart, so it is rejected with `E-NAME-DUP`. That is why the module names in this book's examples differ slightly from their op names.

3.6 Words are a budget

The words the language gives meaning to (keywords) form a closed list. Adding a new word grows the language, so nothing that existing words can express gets a new one. Roughly, they fall into these groups.

Group	Words
Declarations	`module use type newtype struct enum trait contract actor state`
Ops	`fn proc export unsafe extern`
Locals and flow	`let var set return if else for while guard match case try break continue expr`
Values	`make true false none be`
Others	`spawn send drop test expect satisfies do end`

Table 3.3 — Groups of core words

`add`, `len`, `neg` and the like are not words but **builtin ops**. They cannot be used as parameter names either, but they occupy the space of names rather than grammar. Removed words (`in`, `loop`, `as`, `to` and so on) are not quietly accepted; `E-VOCAB-REMOVED` tells you what to use instead.

3.7 The boundaries of the `expr` island

What the island allows fits in one table. `*` and `/` bind tighter than `+` and `-`, comparisons (`eq`, `lt` and so on) come below them, and `and` binds tighter than `or`. Bitwise operations, remainder, minimum and the like are written in prefix form even inside the island. The precedence of bitwise operations differs between languages, so putting them in the island would not make them easier to read — it would add something to look up.

Comparisons cannot be chained.

examples/ch03/chain.low

```
module chain .
rem expect: E-EXPR-CHAIN

fn between input a u32 . input b u32 . input c u32 . output bool .
do
  return expr a lt b lt c .
end
```

Output

```
$ lowentc --check chain.low
chain.low:6:0 E-EXPR-CHAIN: comparisons do not chain in an `expr` island — `a lt b lt c`
 reads like mathematics but means `(a lt b) lt c`, which compares a bool with a number.
Say what you mean: `expr (a lt b) and (b lt c)`
```

Mathematical `a < b < c` means “a is less than b and b is less than c”, but in many languages that spelling reads as `(a < b) < c`. A spelling that means one thing to people and another to the machine is a place where code is silently wrong. As the diagnostic suggests, write `expr (a lt b) and (b lt c)`.

3.8 Clause order in an op head

An op head is a name followed by **clauses**. Each clause is closed by a full stop, and clauses have one fixed order.

Order	Clause	Why here
1	`satisfies` · `lowdoc`	Says **what** the op is first
2	`vector` · `priority`	The character of the op as a whole
3	`comptime` inputs	Later input and output types use these names
4	Capability and region inputs (`cap` … · `region` …)	Who allowed it is visible before what it receives
5	`using`	Which allocator the following data uses
6	Data inputs	The values the op receives
7	`output`	The output type may use the inputs' type parameters
8	`effects`	**What it does** with the capabilities received above
9	`link` · `variadic` · `asm`	Places that connect to the outside (C, machine code)
10	`access` · `parallel` · `reduce`	How it runs split up
11	`requires` → `ensures` → `errors` → `tests`	Input conditions, output promises, failure, tests

Table 3.4 — Clause order in an op head (front to back)

You need not memorise every clause; most ops use four or five of them. What to remember is the principle — clauses are placed so that **earlier ones are used by later ones**. Type parameters before inputs, capabilities before data, inputs before the output, capabilities before effects, effects before contracts.

```
proc copy_upper
  input out cap io .            rem capability input
  input src slice u8 .          rem data input
  output u64 .
  effects io .
  requires gt (len src) 0 .
do
  return write_out out 1 src .
end
```

Written on one line or one clause per line, the meaning is the same (newlines are whitespace). This book writes short heads on one line and heads with contracts one clause per line.

> **In practice. When the output came first**
>
> This order was not settled in one go. For a while the order put `output` at the very front of the head — on the grounds that what an op returns should be seen first. But in generic ops whose output type uses an input's type parameter, that meant a name being used before it was declared, and capabilities ended up far from their effects. So the order was redrawn to follow how clauses use one another, and then fixed. The compiler enforces it with a rank table.

3.9 Common mistakes

Surface mistakes mostly come from **habits carried over from other languages**. Some diagnostics point somewhere unexpected, so it helps to remember the shapes too.

Counter-example. Writing comments with `//`

examples/ch03/mistake_slash.low

```
module mistake_slash .
rem expect: E-RETURN-PARTIAL

fn twice input a u64 . output u64 .
do
  rem ✘ `//` is not a comment marker --- this line and the next become one form ending a
t a single stop
  // double it
  return mul a 2 .
end
```

Output

```
$ lowentc --check mistake_slash.low
mistake_slash.low:4:0 E-RETURN-PARTIAL: this op says it OUTPUTS a value, but some path t
hrough its body reaches the end without a `return`. Until now the tool quietly returned
0 there — a value that appears NOWHERE in your source (RFC-0019 G-TOTAL). Give every pat
h a `return`, or say `output void` if it really produces nothing. An exhaustive `match`
whose every arm returns counts as returning
mistake_slash.low:7:0 E-VOCAB-REMOVED: `//` is not a comment here — this language has ne
ver had it. A comment starts with `rem` (to the end of the line) or `note <tag>` … `<tag
>` (several lines). Until today `//` slipped through to lowering and was reported as an
unsupported FEATURE, which sent the reader looking for a missing capability instead of a
 wrong spelling
```

Comments in this language are `rem` (line comment) and `note WHY … WHY` (multi-line), nothing else. `//` is not a comment, so the compiler reads it as the start of a form, and a form runs **until a stop appears**. `// double it` and the next line's `return mul a 2` therefore become one form that ends at a single stop, and the `return` is swallowed inside it. The diagnostic says "some path does not return", but the cause is the comment. Comments open with a word to keep symbols down — `//`, `#` and `--` differ from language to language, so one word was chosen. The fix: `rem double it`.

Counter-example. Reading a field with a glued dot

examples/ch03/mistake_glued.low

```
module mistake_glued .
rem expect: E-FIELD-GLUED

struct point do
  x u64 .
  y u64 .
end
```

```
fn getx input p point . output u64 .
do
  rem ✘ reads a field with a glued dot --- fields are read with `field p x`
  return p.x .
end
```

Output

```
$ lowentc --check mistake_glued.low
mistake_glued.low:12:10 E-FIELD-GLUED: glued-dot field access is gone — write `(field <v
alue> <name>…)` instead. One meaning gets one spelling: the dot still means module quali
fication (`mod.name`), a variant name (`err.too_short`) and a type-associated declaratio
n (`fn pt.twice`), so a fourth meaning made the same letters mean four things. `field` c
hains: `(field o i z)`
```

`p.x` is everyday C or Python, but here it is `E-FIELD-GLUED`. The dot already serves as a **path to a name** (module `allocs.bump_bytes`, variant `color.red`); if it also looked inside values, every `a.b` would need working out. Read a field with `field p x` (chapter 10).

Counter-example. Writing the head's clauses in any order

examples/ch03/mistake_order.low

```
module mistake_order .
rem expect: E-CLAUSE-ORDER

rem ✘ `output` comes before the input --- clauses have one order
fn twice output u64 . input a u64 .
do
  return mul a 2 .
end
```

Output

```
$ lowentc --check mistake_order.low
mistake_order.low:5:23 E-CLAUSE-ORDER: a data input comes after `output`. An op header h
as ONE order: `satisfies`/`lowdoc` · `vector`/`priority` · `comptime` inputs · capabilit
y/region inputs · `using` · data inputs · `output` · `effects` · `link`/`variadic` · `as
m` · `access`/`inplace`/`invalidates`/`parallel`/`reduce` · `requires` · `ensures` · `er
rors` · `tests` (`--fmt` moves the non-input clauses for you; inputs are call positions,
 so reorder those and their call sites yourself)
```

Clauses have exactly one order (Table 3.4). If the order were free, the same head could be written many ways and readers would have to hunt for each clause. The diagnostic says what came after what, and `lowentc --fmt` puts the order right.

Counter-example. Opening an `if` body without `do`

examples/ch03/mistake_ifdo.low

```
module mistake_ifdo .
rem expect: E-TOPLEVEL
```

```
fn pick input a u64 . output u64 .
do
  rem ✘ the body of `if` was not opened with `do`
  if gt a 3 .
    return 1 .
  end
  return 0 .
end
```

Output

```
$ lowentc --check mistake_ifdo.low
mistake_ifdo.low:10:0 E-TOPLEVEL: this is a STATEMENT, and it sits outside every op body
 — an `end` above it closed the op earlier than you meant. The usual cause is a control
head written without `do`: `if <cond> .` alone takes the ONE statement that follows as i
ts body, so the `end` written for the `if` ends the OP instead. Write the body as `if <c
ond> . do … end` whenever it holds more than one statement
mistake_ifdo.low:4:0 E-RETURN-PARTIAL: this op says it OUTPUTS a value, but some path th
rough its body reaches the end without a `return`. Until now the tool quietly returned 0
 there — a value that appears NOWHERE in your source (RFC-0019 G-TOTAL). Give every path
 a `return`, or say `output void` if it really produces nothing. An exhaustive `match` w
hose every arm returns counts as returning
```

Bodies are not opened by indentation as in Python. Without `do`, the `if` form ends at the stop after the condition, and the `end` below closes **the op's body** rather than the `if`. The remaining `return 0 .` then falls outside any declaration, which gives `E-TOPLEVEL` (something other than a declaration at the top level), and because the body closed early, the return paths go wrong too (`E-RETURN-PARTIAL`). When you see those two together, suspect a missing `do`. The fix: `if gt a 3 . do`.

Counter-example. Writing a string in single quotes

examples/ch03/mistake_quote.low

```
module mistake_quote .
rem expect: E-CHAR-WIDTH

fn size output u64 .
do
  rem ✘ single quotes hold one character --- strings use double quotes, "hi"
  return len 'hi' .
end
```

Output

```
$ lowentc --check mistake_quote.low
mistake_quote.low:7:0 E-CHAR-WIDTH: this character does not fit ONE unit of the prefix y
ou chose. `u8'…'` holds one BYTE (a Hangul syllable is three in UTF-8), `u'…'` one UTF-1
6 code unit (a non-BMP character is a surrogate PAIR), `U'…'` one code point. Widen the
prefix, or use a string — the tool will not silently keep the first unit and call it the
 character
```

Single quotes are the literal for **one character** (`'a'` is the byte 97). Two characters do not fit in one unit, so you get `E-CHAR-WIDTH`. Strings always use double quotes: `len "hi"` is 2. The two kinds of quote mean different things because “one character” and “several bytes” are different types (Table 3.1).

A common misconception. A number with a leading zero is octal

In C, `0755` is octal 493, but Lowent has no octal notation.

examples/ch03/octal.low

```
module octal .
rem run: perm

fn perm output u64 .
do
  rem a leading 0 means nothing --- this is seven hundred fifty-five
  return 0755 .
end
```

Output

```
$ lowentc --run perm octal.low
perm() = 755
```

It was removed because the same characters mean different numbers in different languages. For values where octal is handy, such as permission bits, write `0x1ED` (hexadecimal) or use bit operations.

3.10 This chapter’s syntax at a glance

Shape	Meaning	Why
`add a (mul b c)`	prefix notation — name first, inner calls in parentheses	no precedence rules to memorise
`… .`	a detached stop closes a form (statement or clause)	newlines never change meaning — break lines anywhere
`do … end`	every block	only one way to open a block
`rem … · note WHY … WHY`	line comment · multi-line comment	no symbol comments (`//`, `#`) — they open with a word
`42 · 0x2A · 0b101010 · 1_000`	integer literals (the position decides the type)	no octal — `0755` is 755
`"hi\n" · 'a'`	a string (several bytes) · one character	the quote itself is the type difference
`true · false · none`	booleans · no value	values are words too
`field p x · method s area`	read a field · call a method	the dot is kept for name paths only

`allocs.bump_bytes` · `color.red`	a name inside a module · a variant name	one dot means “that name inside this name”
`expr a + b * c`	an infix island — arithmetic, comparisons, `and`/`or` only	long arithmetic reads easily; same meaning as prefix
Clause order in an op head	Table 3.4	each clause comes before the ones that use it
`0b1010` · `1_000_000` · `0x1.8p1` · `u'가'` · `U"…"`	binary · digit separator · hex float · prefixed char and string	no octal — `0755` is 755
`text DOC … DOC` · `note END … END`	multi-line string (escapes not unescaped) · multi-line comment	long text goes in as written

Table 3.5 — Surface rules — shape · meaning · why it looks this way

Recap

A free-standing full stop closes a form, and a newline is whitespace. Expressions are prefix with the name first, and infix is written only inside `expr` islands. A block is always `do … end`. Names contain no dots, and shadowing is forbidden. A literal that does not fit its place’s type is rejected. The clauses of an op head have one order, arranged so that earlier clauses are used by later ones.

Part II — Values and flow

4 Numbers — fixed-width integers and floating point

In this chapter

4.1 Integer types name their width 52
4.2 Widening is automatic, narrowing is written 52
4.3 Overflow stops 54
4.4 Division and remainder 55
4.5 Booleans are not numbers 56
4.6 Bitwise operations 57
4.7 Floating point 58
4.8 Common mistakes 60
4.9 This chapter's syntax at a glance 65

What to know first

chapter 2, A first program · the VM and native code must agree
chapter 3, The surface · a literal that does not fit its place's type is rejected

Looking back

What happened to `let x u8 be 300 .` in chapter 3? How does C handle the same thing?

> **A.** It was rejected with `E-TYPE-WIDTH`, because 300 is outside the `u8` range 0 … 255. C silently truncates it to 44, so the value in the source and the actual value differ. This chapter looks at how the same principle applies when a **computed result**, not a literal, exceeds its width.

The need for this chapter, and its context

Integer defects are among the oldest and most common in systems programs. Values overflow into small numbers, negative numbers turn into huge positive ones, and division by zero becomes undefined behaviour. Most of it happens **silently** and surfaces much later in an unrelated place. Numbers open Part II because all computation happens on them, and because they show in the simplest form how Lowent removes "silently" — by making you write the chosen treatment as a name.

By the end of this chapter

You will learn that integer types are named by sign and width and have the same size everywhere. You will pick up the rule that widening is automatic and narrowing must be written, that overflow and division by zero stop execution, and how to choose **by name** to wrap, saturate or get the failure back as a value instead. You will also see that booleans are not numbers, and the behaviour of bitwise operations and floating point.

The questions this chapter answers

1. If `narrow_wrap` does what C does, why isn't wrapping the default?
2. Why can't bitwise operations be written infix inside an `expr` island?

4.1 Integer types name their width

An integer type's name is its sign and bit width. `u` means unsigned, `i` means signed.

Type	Width	Range
`u8` · `u16` · `u32` · `u64`	8 · 16 · 32 · 64	$0 \dots 2^N-1$
`i8` · `i16` · `i32` · `i64`	8 · 16 · 32 · 64	$-2^{N-1} \dots 2^{N-1}-1$
`usize` · `isize`	address width	set by the execution environment

Table 4.1 — Integer types and their ranges

C's `int` can have different sizes on different machines, but Lowent's `u32` is 32 bits everywhere. Negative numbers use two's complement. `usize` is a **different type** from `u64` even on a machine where they have the same width, and the two do not convert automatically — so that places mixing address counts with plain numbers show in the source.

4.2 Widening is automatic, narrowing is written

A conversion that loses no value is called **widening**. Passing a `u8` where a `u32` is expected keeps the value, so the compiler does it for you. Even with mixed signs, arithmetic is allowed if a value-preserving widening exists — every `u8` fits in an `i16`.

examples/ch04/widths.low

```
module widths .
rem run: mix 200 -100
rem run: shrink 200
rem trap: shrink 300
rem run: shrink_sat 300
rem run: shrink_wrap 300

fn mix input a u8 . input b i16 . output i16 .
do
  return add a b .
end

fn shrink input x u64 . output u8 .
do
  return narrow u8 x .
end

fn shrink_sat input x u64 . output u8 .
do
  return narrow_sat u8 x .
end

fn shrink_wrap input x u64 . output u8 .
```

```
do
  return narrow_wrap u8 x .
end
```

Output

```
$ lowentc --run mix widths.low 200 -100
mix(200, -100) = 100
$ lowentc --run shrink widths.low 200
shrink(200) = 200
$ lowentc --run shrink_sat widths.low 300
shrink_sat(300) = 255
$ lowentc --run shrink_wrap widths.low 300
shrink_wrap(300) = 44
$ lowentc --run shrink widths.low 300
== ir diagnostics (1) ==
0:0 E-VM-CAST: value does not fit the target width (use narrow_wrap / narrow_sat / narro
w_try)
```

`mix` adds a `u8` and an `i16`, and the result has the wider type, `i16`. To make the widening visible, write `widen u64 x`. An `i32` and a `u32` of the same width cannot be mixed, because neither fits entirely in the other.

examples/ch04/sign_bad.low

```
module sign_bad .
rem expect: E-TYPE-SIGN

fn total input a i32 . input b u32 . output i32 .
do
  return add a b .
end
```

Output

```
$ lowentc --check sign_bad.low
sign_bad.low:6:0 E-TYPE-SIGN: sign mismatch: no value-preserving widening exists (widen
both to a strictly wider signed type, or bitcast_sign)
```

A conversion that can lose a value is **narrowing**, and it must be written. `shrink` in the same file narrows a `u64` to a `u8` with `narrow u8 x`. 200 fits and comes out unchanged, but 300 does not, so execution stops (`E-VM-CAST`). If you do not want it to stop, choose the treatment you want by name. `narrow_sat` stops at the end value 255, and `narrow_wrap` wraps like C and gives 44. `narrow_try` gives `none` when the value does not fit.

> **Q. If `narrow_wrap` does what C does, why isn't wrapping the default?**
>
> **A.** To leave in the source whether wrapping was **intended or a mistake**. If wrapping were the default, someone reading a place where `narrow u8 x` gave 44 could not tell whether that was the wanted value. With stopping as the default, an author who wanted wrapping **wrote** `narrow_wrap`. The name written is the record of the author's intent.

4.3 Overflow stops

Arithmetic happens **at the declared width**. If the result does not fit the width, execution stops (a trap).

examples/ch04/overflow.low

```
module overflow .
rem run: bump 254
rem trap: bump 255
rem run: bump_wrap 255
rem run: bump_sat 255

fn bump input a u8 . output u8 .
do
  return add a 1 .
end

fn bump_wrap input a u8 . output u8 .
do
  return wrap_add a 1 .
end

fn bump_sat input a u8 . output u8 .
do
  return sat_add a 1 .
end
```

Output

```
$ lowentc --run bump overflow.low 254
bump(254) = 255
$ lowentc --run bump_wrap overflow.low 255
bump_wrap(255) = 0
$ lowentc --run bump_sat overflow.low 255
bump_sat(255) = 255
$ lowentc --run bump overflow.low 255
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
```

`bump 254` fits as 255, but `bump 255` would have to be 256, so the VM stops with `E-VM-OVERFLOW`. Native code stops at the same place with a non-zero exit code (the verification script checks that both stop). As with widening, you change the treatment by choosing a name.

Treatment	Ops	On overflow
Stop (default)	`add` · `sub` · `mul`	Execution stops
Wrap	`wrap_add` · `wrap_sub` · `wrap_mul`	Wraps around within the width
Saturate	`sat_add` · `sat_sub` · `sat_mul`	Stops at the end value of the width
As a value	`chk_add` · `chk_sub` · `chk_mul`	Reports it as an `option`

Table 4.2 — Choosing the treatment of overflow by name

The “as a value” family turns overflow into an event the program can handle.

examples/ch04/chk.low

```
module chk .
rem run: safe_bump 255
rem run: safe_bump 7

fn safe_bump input a u8 . output u64 .
do
  let r option u8 be chk_add a 1 .
  guard is_some r . else return 999 .
  return widen u64 (some_value r) .
end
```

Output

```
$ lowentc --run safe_bump chk.low 255
safe_bump(255) = 999
$ lowentc --run safe_bump chk.low 7
safe_bump(7) = 8
```

`chk_add a 1` gives an `option u8`. On overflow it is `none`, and `guard is_some r . else …` handles that case first. You will meet this shape again in chapter 11.

> **A common misconception. Checking for overflow makes code slow**
>
> It would if every addition kept its check. But where a contract or a range type proves the range of a value, the check is removed at translation time (chapter 14). Places with an obvious range, such as loop counters, are cleaned up by the compiler’s interval analysis. The checks that remain are at places that **really can overflow**, and there the check is not a cost but the value that stops a defect.

4.4 Division and remainder

Dividing an integer by zero stops. Signed division truncates towards zero, and the sign of the remainder `mod` follows the **divisor**.

examples/ch04/divide.low

```
module divide .
rem run: quot -7 2
rem run: modulo -7 2
rem trap: quot 1 0
rem run: safe_div 7 0
rem run: safe_div 7 2

fn quot input a i32 . input b i32 . output i32 .
do
  return div a b .
end

fn modulo input a i32 . input b i32 . output i32 .
do
  return mod a b .
end
```

```
fn safe_div input a u32 . input b u32 . output u32 .
do
  let nz option nonzero u32 be nonzero_of b .
  guard is_some nz . else return 0 .
  return div_nz a (some_value nz) .
end
```

Output

```
$ lowentc --run quot divide.low -7 2
quot(-7, 2) = -3
$ lowentc --run modulo divide.low -7 2
modulo(-7, 2) = 1
$ lowentc --run safe_div divide.low 7 0
safe_div(7, 0) = 0
$ lowentc --run safe_div divide.low 7 2
safe_div(7, 2) = 3
$ lowentc --run quot divide.low 1 0
== ir diagnostics (1) ==
0:0 E-VM-DIV0: divide by zero
```

`quot -7 2` is −3 and `modulo -7 2` is 1. Because the remainder follows the divisor's sign, `mod h n` for a positive `n` is always at least 0 and less than `n`. That property is why the bounds check in `index slots (mod h n)` disappears when a hash table picks a slot, and it is proven in Coq (chapter 40).

The only case where signed division overflows is `MIN / -1`, and that stops too. In C it is undefined behaviour.

There is also a way to check once that a divisor is not zero and carry that fact around. `nonzero_of b` gives an `option nonzero u32`, and the value inside can be passed to `div_nz`. No zero check remains at the `div_nz` site. The check was not removed but **moved to one place, with its result carried in the type**.

4.5 Booleans are not numbers

`bool` has two values, `true` and `false`. An integer in a condition is rejected.

examples/ch04/cond_bad.low

```
module cond_bad .
rem expect: E-TYPE-COND

fn nonzero_flag input n u32 . output u8 .
do
  if n . do return 1 . end
  return 0 .
end
```

Output

```
$ lowentc --check cond_bad.low
cond_bad.low:6:0 E-TYPE-COND: a condition must be bool, not a number — this language has
 no truthiness (D12: no implicit int↔bool). Say what the test is: `ne x 0`
```

C's `if (n)` reads as "n is not zero", but the reader has to guess from context whether it means "n exists" or "n is true". Lowent makes you write what you are asking — `if gt n 0 ..`. Conversely, a `bool` cannot be added like a number. `and`, `or` and `not` take only booleans, and `and` and `or` skip the right-hand side when the left-hand side already decides the answer.

4.6 Bitwise operations

Bitwise operations treat a value as a sequence of bits. They use words instead of symbols.

examples/ch04/bits.low

```
module bits .
rem run: mask 12 10
rem run: flip 12
rem run: spin 129 1
rem run: s_shr -8 1
rem run: u_shr 4294967288 1
rem run: ones 7

rem 12 = 0000_1100 · 10 = 0000_1010
fn mask input a u8 . input b u8 . output u8 . do return bit_and a b . end
fn flip input a u8 . output u8 . do return bit_not a . end
fn spin input a u8 . input n u8 . output u8 . do return rotl a n . end
fn s_shr input a i32 . input n i32 . output i32 . do return shr a n . end
fn u_shr input a u32 . input n u32 . output u32 . do return shr a n . end
fn ones input a u8 . output u8 . do return count_ones a . end
```

Output

```
$ lowentc --run mask bits.low 12 10
mask(12, 10) = 8
$ lowentc --run flip bits.low 12
flip(12) = 243
$ lowentc --run spin bits.low 129 1
spin(129, 1) = 3
$ lowentc --run s_shr bits.low -8 1
s_shr(-8, 1) = -4
$ lowentc --run u_shr bits.low 4294967288 1
u_shr(4294967288, 1) = 2147483644
$ lowentc --run ones bits.low 7
ones(7) = 3
```

- `bit_and` · `bit_or` · `bit_xor` · `bit_not` — bitwise logic. `flip 12` is 243 because the type is `u8`. The answer of a bitwise operation is always tied to the width of its type.
- `shl` and `shr` shift; `rotl` and `rotr` rotate. `spin 129 1` is 3 because the top 1 came back in at the bottom; with `shl` that 1 would have been discarded.
- `shr` means different things depending on sign. −8 and 4294967288 are the same bit pattern in 32 bits, but the signed one shifts in the sign bit and becomes −4, while the unsigned one shifts in 0 and becomes 2147483644.
- `count_ones` · `leading_zeros` · `trailing_zeros` count bits, and `byte_swap` reverses byte order.
- `clmul_lo` · `clmul_hi` are **carry-less multiplication** — a product whose addition is xor, with no carries. Cryptography and checksums (GF(2^128), CRC) stand on it. The answer is 128 bits, so it comes back as two words. When the machine has the instruction the processor lowers to it

(`lowentc --hw pclmul`, or `auto`), and when it does not, to the same computation in plain code — **one meaning**, two speeds.

Shifting by at least the width of the type stops. In C that is undefined behaviour. If you want the shift amount wrapped into the width, use `wrap_shl` and `wrap_shr`.

> **Q. Why can't bitwise operations be written infix inside an `expr` island?**
>
> **A.** In C, `a & b == c` groups as `a & (b == c)`. It is a trap even for long-time users, and languages disagree on bitwise precedence. Putting them in the island would add something to look up instead of making them easier to read. Written prefix, the parentheses state the answer and nothing can be misread.

4.7 Floating point

`f32` and `f64` are the IEEE 754 binary 32- and 64-bit formats. Operations happen at the width of their operands, and integers and floating-point values do not convert to each other automatically.

examples/ch04/floats.low

```
module floats .
rem run: third 1.0
rem run: hyp 3.0 4.0

fn third input x f64 . output f64 .
do
  return div x 3.0 .
end

fn hyp input a f64 . input b f64 . output f64 .
do
  return sqrt (add (mul a a) (mul b b)) .
end
```

Output

```
$ lowentc --run third floats.low 1.0
third(1.0) = 0.333333
$ lowentc --run hyp floats.low 3.0 4.0
hyp(3.0, 4.0) = 5.0
```

The VM shows floating-point results briefly (`0.333333`). Dividing a floating-point value by zero does not stop; it becomes infinity as IEEE 754 prescribes. Transcendental functions such as `sqrt`, `sin` and `exp` are floating-point only and available only on machines with an operating system. The exact last bit of their results is up to the machine and its maths library.

> **A common misconception. Floating-point values can be compared with `eq` too**
>
> 0.1 plus 0.2 does not have the same bits as 0.3. Floating-point equality usually has to be asked as "close enough", and the standard library's `math` module provides that question (`close`)

> (chapter 32). Rounding, NaN and −0 are also properties the proofs of this language do not cover (chapter 50).

`f32` and the size types follow the same rules.

examples/ch04/sizes.low

```
module sizes .
rem run: step 5 -8
rem run: third32 1.0
rem run: same_third 1.0 1.0

rem usize and isize have the machine's address width (64 bits on a 64-bit machine); use
them to exchange lengths and indexes with C
fn step input n usize . input k isize . output isize .
  requires le n 1000 .
  requires ge k -1000 .
  requires le k 1000 .
do
  return add (cast isize n) k .
end

rem f32 is a 32-bit float: half the memory, about seven significant digits
fn third32 input x f32 . output f32 . do
  return div x 3.0 .
end

rem even 1/3 is a different number in f32 and f64; widened and compared, they are not eq
ual
fn same_third input x f32 . input y f64 . output bool . do
  let a f32 be div x 3.0 .
  let b f64 be div y 3.0 .
  return eq (widen f64 a) b .
end
```

Output

```
$ lowentc --run step sizes.low 5 -8
step(5, -8) = -3
$ lowentc --run third32 sizes.low 1.0
third32(1.0) = 0.333333
$ lowentc --run same_third sizes.low 1.0 1.0
same_third(1.0, 1.0) = 0
```

- `usize` and `isize` are unsigned and signed integers with the machine's address width. Use them to exchange lengths and indexes with C's `size_t` and `ptrdiff_t` (chapter 29). Their width may differ between machines, so use `u64` and `i64` for fixed-width arithmetic. Moving to a type of the other signedness goes through a named operation such as `cast isize n`.
- `f32` takes half the memory and has about seven significant digits. Use it where quantity matters more than precision, as in large arrays or graphics.
- `same_third` computes the same 1/3 as `f32` and as `f64`, widens, and compares. The answer is false (0). Even though the VM shows both briefly as `0.333333`, they are different numbers.

4.8 Common mistakes

Many number mistakes pass compilation and only show up **while running**. Where other languages would quietly produce a wrong value, Lowent stops — and the place it stops is the place to fix.

> **Counter-example. Finding the middle by adding first, then halving**

examples/ch04/mistake_midpoint.low

```
module mistake_midpoint .
rem trap: mid 200 100

fn mid input a u8 . input b u8 . output u8 .
do
  rem ✘ adding first gives 300, which does not fit in u8
  return div (add a b) 2 .
end
```

Output

```
$ lowentc --run mid mistake_midpoint.low 200 100
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
```

`200 + 100` is 300, and `u8` only goes up to 255, so it overflows before the division. This is the very bug that lurked for decades in `(lo + hi) / 2` inside binary searches. C would quietly give 22, half of the wrapped value 44. The fix is to compute **at a wider width**.

examples/ch04/midpoint_fixed.low

```
module midpoint_fixed .
rem run: mid 200 100
rem run: mid 255 255

fn mid input a u8 . input b u8 . output u8 .
do
  rem add one width up --- u16 holds up to 510
  let both u16 be add (widen u16 a) (widen u16 b) .
  rem half of it fits in u8 again; we know that, and narrow checks it
  return narrow u8 (div both 2) .
end
```

Output

```
$ lowentc --run mid midpoint_fixed.low 200 100
mid(200, 100) = 150
$ lowentc --run mid midpoint_fixed.low 255 255
mid(255, 255) = 255
```

Widen to `u16`, add, divide, then `narrow u8` back down. The middle always fits in `u8`, so the narrowing never stops — and if the computation is ever changed by mistake, it stops right there and tells you. (When `lo <= hi` is guaranteed, `add lo (div (sub hi lo) 2)` does not overflow either.)

Counter-example. Subtracting a larger number from an unsigned one

examples/ch04/mistake_usub.low

```
module mistake_usub .
rem trap: gap 3 5

fn gap input a u64 . input b u64 . output u64 .
do
  rem ✘ unsigned numbers cannot go below 0 --- 3 − 5 is not negative, it stops
  return sub a b .
end
```

Output

```
$ lowentc --run gap mistake_usub.low 3 5
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
```

`u64` has nothing below 0. `3 - 5` is not −2 but an **overflow**, so it stops (C would give 18446744073709551614). If you want the size of the difference, subtract the smaller from the larger — `if ge a b . do return sub a b . end return sub b a .` — and if a negative result is meaningful, compute in `i64` from the start.

Counter-example. Giving a loop counter too narrow a type

examples/ch04/mistake_narrowloop.low

```
module mistake_narrowloop .
rem expect: E-TYPE-WIDTH

fn tally output u64 .
do
  rem ✘ a u8 loop counter has nothing after 255 --- the 256th `add i 1` stops
  var i u8 be 0 .
  var n u64 be 0 .
  while lt i 256 . do
    set n (add n 1) .
    set i (add i 1) .
  end
  return n .
end
```

Output

```
$ lowentc --check mistake_narrowloop.low
mistake_narrowloop.low:9:0 E-TYPE-WIDTH: this comparison holds a value that the other si
de's type cannot hold, so the answer is the same for every input — the comparison decide
s nothing. A name declared `u8` is at most 255, `u16` at most 65535, `i8` at most 127; a
 literal beyond that is out of range exactly as it would be in an assignment, where the
tool has always refused it. Widen the name's type, or compare against a value the type c
an reach. (This is how a loop written `while lt i 256 .` over a `u8` counter never ends)
```

A `u8` can never reach 256, so `lt i 256` is always true. In C, `i` would wrap from 255 back to 0 and the loop would **never end**. Lowent refuses the comparison at translation time with `E-TYPE-WIDTH`: comparing against a value the type cannot hold gives the same answer for every input, so it decides nothing. Give a loop counter a type wider than the largest value it counts to (`u64`).

Counter-example. Doing float arithmetic with an integer literal

examples/ch04/mistake_floatint.low

```
module mistake_floatint .
rem expect: E-TYPE-MIX

fn half input x f64 . output f64 .
do
  rem ✘ divides an f64 by the integer literal 2 --- write 2.0 where a float is expected
  return div x 2 .
end
```

Output

```
$ lowentc --check mistake_floatint.low
mistake_floatint.low:7:0 E-TYPE-MIX: a floating-point value and an INTEGER literal are m
ixed in one operation. Floating point and integers do not convert implicitly (§6.2.5), s
o write the literal as a float (`2.0`) — or convert the other side. It used to pass `--c
heck` and stop at run time (`E-VM-TYPE: arithmetic needs ints`)
```

Integers and floats never convert into each other automatically. To divide an `f64`, write the literal as a float too: `2.0`. Mixing them is rejected at translation time with `E-TYPE-MIX`; it used to stop **while running** with `E-VM-TYPE`.

Counter-example. Testing with the largest `u64` passed as a `--run` argument

examples/ch04/mistake_runmax.low

```
module mistake_runmax .
rem trap: echo 18446744073709551615
rem test

rem ✘ giving the upper half of u64 as a `--run` argument is refused at the boundary: the
 interval analysis believes u64 tops out at the i64 maximum
fn echo input n u64 . output u64 .
do
  return n .
end

rem write boundary values as literals inside a test
test largest_u64
do
  let m u64 be 18446744073709551615 .
  expect gt m 9223372036854775807 .
  expect eq (div m 2) 9223372036854775807 .
end
```

Output

```
$ lowentc --run echo mistake_runmax.low 18446744073709551615
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: a parameter's TYPE range was violated at the program boundary — the v
alue does not fit the declared type (the interval analysis TRUSTS the type and deletes c
hecks on that basis; the boundary is what makes that trust true)
$ lowentc --test mistake_runmax.low
  [PASS] largest_u64
== tests: 1 run, 1 passed, 0 FAILED ==
```

`echo 18446744073709551615` is refused at the boundary (`E-VM-CONTRACT`). Until 2026-09-16 it was quietly cut down to 9223372036854775807 — a different number, and both back ends did it, so comparing them did not reveal it. It is now refused instead of cut. The reason is that the interval analysis records the upper bound of `u64` as the largest `i64`; a value above that makes the analysis unsound. Literals inside the program are whole, so write boundary-value tests there, as `largest_u64` does, and results now print unsigned.

A common misconception. `f32` and `f64` hold the same number

examples/ch04/mistake_f32literal.low

```
module mistake_f32literal .
rem run: stored
rem run: computed

rem ✘ an f32 holding a literal as is; in this edition it is not rounded to 32 bits, so i
t compares equal to the f64 0.1
fn stored output bool . do
  let a f32 be 0.1 .
  let b f64 be 0.1 .
  return eq (widen f64 a) b .
end

rem an f32 that went through one operation (0.1 + 0) is rounded to 32 bits and compares
unequal
fn computed output bool . do
  let a f32 be 0.1 .
  let c f32 be add a 0.0 .
  let b f64 be 0.1 .
  return eq (widen f64 c) b .
end
```

Output

```
$ lowentc --run stored mistake_f32literal.low
stored() = 0
$ lowentc --run computed mistake_f32literal.low
computed() = 0
```

Both `stored` and `computed` are false. The 0.1 held in an `f32` place is rounded to 32 bits, so widening it still does not equal the `f64` 0.1. Until 2026-09-16 a literal (or an argument passed with `--run`) was not rounded, and `stored` was true — the same type held values of different

precision depending on where they came from. Choose the width deliberately, and write `widen` or `narrow` when comparing two widths.

A common misconception. `div 7 2` is 3.5

Integer division keeps only the quotient and drops the fraction.

examples/ch04/intdiv.low

```
module intdiv .
rem run: avg2 3 4
rem run: avg_f 3.0 4.0

fn avg2 input a u64 . input b u64 . output u64 .
do
  rem integer division drops the fraction --- 7 / 2 is 3
  return div (add a b) 2 .
end

fn avg_f input a f64 . input b f64 . output f64 .
do
  rem if you need the fraction, compute in floating point from the start
  return div (add a b) 2.0 .
end
```

Output

```
$ lowentc --run avg2 intdiv.low 3 4
avg2(3, 4) = 3
$ lowentc --run avg_f intdiv.low 3.0 4.0
avg_f(3.0, 4.0) = 3.5
```

`avg2 3 4` is 3. If you need the fraction, take the inputs as floats and divide by `2.0`. If you need rounding, **write** it — add `div d 2` before dividing, for example — because the language never rounds for you.

The floating-point remainder gets its own name, `fmod`, separate from the integer `mod`.

examples/ch04/floatmod.low

```
module floatmod .
rem run: rest
rem run: whole_rest 7 2

rem the floating-point remainder is `fmod` --- a different name from the integer `mod`
fn rest output f64 .
do
  return fmod 7.5 2.0 .
end

fn whole_rest input a u64 . input b u64 . output u64 .
  requires gt b 0 .
do
  return mod a b .
end
```

Output

```
$ lowentc --run rest floatmod.low
rest() = 1.5
$ lowentc --run whole_rest floatmod.low 7 2
whole_rest(7, 2) = 1
```

`fmod 7.5 2.0` is 1.5 and `mod 7 2` is 1. Not loading two kinds onto one name is the same reason as for `div` — what is computed shows in the name.

4.9 This chapter's syntax at a glance

Shape	Meaning	Why
u8 ⋯ u64 · i8 ⋯ i64 · usize · isize	integers with the width in the name	sizes never change from machine to machine
f32 · f64	IEEE 754 floating point	never mixed with integers — the literal is `2.0` too
widen u64 x	widening (loses nothing)	happens automatically, but can be written to make it visible
narrow u8 x	narrowing — stops if it does not fit	so values never change silently
narrow_sat · narrow_wrap · narrow_try	clamp to the end value · wrap · as an `option`	pick the outcome by name
add · sub · mul	arithmetic that stops on overflow	overflow is a bug by default
wrap_add · sat_add · chk_add	wrap · saturate · `option`	the intended outcome stays in the source
div · mod	quotient (towards zero) · remainder (sign of the divisor)	division by zero and `MIN / -1` stop
nonzero_of b · div_nz	check "not zero" once and carry it in the type	moves the check to one place
bit_and · bit_or · bit_xor · bit_not	bitwise logic	no symbol-precedence traps
shl · shr · rotl · rotr	shift · rotate	shifting by the width or more stops
clmul_lo · clmul_hi	carry-less multiply — two words out	the machine instruction when there is one, the computation when there is not
count_ones · leading_zeros · trailing_zeros · byte_swap	count bits · reverse bytes	common jobs get one name
true · false · and · or · not	booleans and their logic	numbers are never used as conditions

`usize` · `isize` · `f32`	address-width integers · 32-bit float	lengths exchanged with C · when quantity matters more than precision

Table 4.3 — Number syntax — shape · meaning · why it looks this way

Recap

Integer types carry sign and width in their names and have the same size everywhere. Widening is automatic and narrowing is written with `narrow`. Overflow, division by zero, out-of-range narrowing and shifting by the width or more all stop, and other treatments (wrap, saturate, as a value) are chosen by names like `wrap_`, `sat_`, `chk_` and `narrow_try`. The sign of `mod` follows the divisor. Booleans are not numbers, bitwise operations are written as words, and width and sign decide their meaning.

5 Ops — `fn` and `proc`

In this chapter

5.1 Two kinds 67
5.2 Purity is decided by observation 68
5.3 Don’t write `effects` on a `fn` 69
5.4 Effects spread to the caller 70
5.5 Parameters and return 70
5.6 Only `neg` is unary 71
5.7 Modifiers 72
5.8 Common mistakes 72
5.9 This chapter’s syntax at a glance 75

What to know first

chapter 3, The surface · clause order in an op head
chapter 4, Numbers · overflow stops and the treatment is chosen by name

Looking back

In chapter 3’s clause-order table, what do `output` and `effects` each come after? Why there?

> **A.** `output` comes after the data inputs, because the output type may use an input’s type parameter. `effects` comes after `output` and before the contracts, because it is where you write what is done with the capabilities received earlier. This chapter covers why the unit of execution that has such a head — the op — splits in two.

The need for this chapter, and its context

The basic unit of a Lowent program is the op, and an op is always either a `fn` or a `proc`. That split is the floor on which the effect system of Part IV, the traits of Part VI and the parallelism of Part VII all stand, because a pure op may have its result remembered, be reordered, or be run in parallel. So the two kinds of op come right after numbers, before locals and flow.

By the end of this chapter

You will learn the difference between `fn` and `proc`, and what it means that purity is **observational** (mutation confined inside the op counts as pure). You will pick up how to receive parameters with `input` clauses and return with `output`, recursion, and the fact that `neg` is the only unary arithmetic op. You will also see what gets rejected when a pure op calls an impure one, and when a `mut` parameter writes back to the caller.

The questions this chapter answers

1 What happens if a `proc` actually performs no effect?

5.1 Two kinds

An **op** (operation) is a unit of execution with a name and a contract. On the surface it always opens with one of two words.

- `fn` — a **pure** op. The same inputs give the same result, and it leaves no trace outside.
- `proc` — an op that may perform effects. Which effects it performs is written in its `effects` clause.

There is no default. Whether it is a `fn` or a `proc` is always written before the name. Builtin operations such as `add` and `len` are ops too — ops the language made in advance, called **builtin ops**.

examples/ch05/kinds.low

```
module kinds .
rem run: add3 1 2 3
rem run: running_total [4,5,6]
rem run: fact 10

fn add3 input a i32 . input b i32 . input c i32 . output i32 .
do
  return add (add a b) c .
end

rem local mutable state stays inside the op, so a fn may use it
fn running_total input xs slice u8 . output u64 .
do
  var total u64 be 0 .
  for x xs do
    set total (add total (widen u64 x)) .
  end
  return total .
end
```

```
fn fact input n u64 . output u64 .
  requires le n 20 .
do
  if le n 1 . do return 1 . end
  return mul n (fact (sub n 1)) .
end
```

Output

```
$ lowentc --run add3 kinds.low 1 2 3
add3(1, 2, 3) = 6
$ lowentc --run running_total kinds.low [4,5,6]
running_total([4,5,6]) = 15
  arg0 (written) = [4,5,6]
$ lowentc --run fact kinds.low 10
fact(10) = 3628800
```

All three ops are `fn`s. `add3` adds three arguments, and `fact` is recursive, calling itself. `fact`'s `requires le n 20 .` is there because 21! exceeds `u64`. Without that contract, `fact 21` would stop at a multiplication (chapter 4). With it, execution stops **on entry**, and where it stopped tells you it was the caller's fault.

5.2 Purity is decided by observation

Look at `running_total` again. Its body has `var total` and changes it with `set`. Yet it is a `fn`. Lowent's purity is **observational**: mutating a local variable confined inside the op cannot be seen from outside, so it counts as pure. Even if memory is written at the machine level, it is not an effect if the caller cannot observe it.

On the other hand, a write the caller can see is an effect. A `fn` that writes through a `mut` parameter is rejected.

examples/ch05/mutparam.low

```
module mutparam .
rem expect: E-EFFECT-PURITY

fn zero_first input xs mut slice u64 . output u64 .
do
  set (index xs 0) 0 .
  return 0 .
end
```

Output

```
$ lowentc --check mutparam.low
mutparam.low:4:0 E-EFFECT-PURITY: a `fn` WRITES through a `mut` parameter — that write i
s visible to the CALLER. A fn is an enforced purity contract (SPEC-003 §27): callers may
 memoise it, reorder it, or elide it. An op that changes caller-owned storage can do non
e of those. Declare it a `proc`. (Local mutation stays pure: storage confined to the op
is not observable — a machine write is not an observable effect. RFC-0057)
```

As the diagnostic explains, a `fn` is an enforced purity contract. A caller may remember a `fn`'s result (memoise), reorder calls, or drop a call whose result is unused. None of that is possible for an op that changes the caller's storage. So it must be a `proc`.

examples/ch05/mutproc.low

```
module mutproc .
rem run: zero_first [7,8,9]

proc zero_first input xs mut slice u8 . output u64 . effects state .
  requires gt (len xs) 0 .
do
  set (index xs 0) 0 .
  return len xs .
end
```

Output

```
$ lowentc --run zero_first mutproc.low [7,8,9]
zero_first([0,8,9]) = 3
  arg0 (written) = [0,8,9]
```

`effects state .` declares that this op changes state outside itself. The VM's `arg0 (written) = [0,8,9]` is the trace — the first element of the slice the caller passed was changed.

> **A common misconception. A `proc` is slower than a `fn`**
>
> The kind has nothing to do with speed. Both go down to C the same way. The difference is in **what the compiler and the caller may do**. Reordering, common-subexpression elimination and memoisation are allowed for a `fn` and not for a `proc`. Writing pure work as a `proc` throws those chances away.

5.3 Don't write `effects` on a `fn`

A `fn`'s effects are already `none`. So adding `effects none .` is rejected.

examples/ch05/redundant.low

```
module redundant .
rem expect: E-EFFECT-REDUNDANT

fn square input a u64 . output u64 . effects none .
do
  return mul a a .
end
```

Output

```
$ lowentc --check redundant.low
redundant.low:4:0 E-EFFECT-REDUNDANT: a `fn` is pure by contract, so `effects none .` sa
ys nothing that the `fn` did not already say — DROP the clause. One meaning must have on
e spelling (SPEC-002 §2.5). (Keep writing `effects` on `proc`: there it NARROWS, and omi
tting it means unrestricted.)
```

Not because it is wrong, but because it would be **two spellings of one meaning**. On a `proc` things are different. A `proc`'s `effects` clause **narrows**. A `proc` without the clause reads as not narrowed — it is assumed able to do input/output, allocation and state. So a `proc` should carry the clause; doing more than it lists is rejected, and effects it lists but never performs are reported.

> **Q. What happens if a `proc` actually performs no effect?**
>
> **A.** Writing `effects none .` on a `proc` is allowed. Older code sometimes keeps an op as a `proc` to emphasise local mutable state or loops. But if no effect is observable, writing it as a `fn` is the shape this language recommends. Purity has to show in the head for callers and the compiler to use it.

5.4 Effects spread to the caller

Calling an op adds its effects to the caller's. When a pure `fn` calls a `proc` that performs `io`, that `fn` is no longer pure.

examples/ch05/calls.low

```
module calls .
rem expect: E-EFFECT-CALC

proc say input out cap io . output u64 . effects io .
do
  return write_out out 1 "hi\n" .
end

fn helper input out cap io . output u64 .
do
  return say out .
end
```

Output

```
$ lowentc --check calls.low
calls.low:10:1 E-EFFECT-CALC: this fn is declared pure but performs `io` — make it a `pr
oc` with `effects …`, or remove the effect
```

`helper` does not print by itself; `say` does. It is still rejected. Thanks to this rule, the head of an op tells you the effects of every op it calls. There is no way to pass an effect along hidden. Which effect pairs with which capability is covered in chapters 15 and 16.

5.5 Parameters and return

Here is one op taken apart. What trips up newcomers most is that **every clause ends with a full stop**. The stop works like the end of a sentence: "this clause is finished".

```
fn   add3   input a i32 .   input b i32 .   input c i32 .   output i32 .
|    |      └─ parameter a has type i32 ─┘                   └ type of the returned value
|    └─ name (used when calling)
└─ kind: fn (pure) or proc (may have effects)
do                          ← the body starts here
```

```
  return add (add a b) c .  ← compute and return; a statement also ends with a stop
end                         ← the body ends, and so does the whole declaration (no stop after it)
```

To call an op, write its name followed by the arguments **separated by spaces**. `add3 1 2 3` is one call, and parentheses are used only to put one call inside another as an argument — `add3 (add3 1 2 3) 4 5`. Parentheses simply mean "this is one value". There is no C-style `add3(1, 2, 3)`: the comma has no meaning in this language, so it is rejected (see «Common mistakes» below).

A parameter is one `input <name> <type> .` clause. For several, repeat the clause. The caller supplies arguments in clause order. The returned value is a single `output <type> .`, and the body returns it with `return <expr> .`.

```
fn compare input a i32 . input b i32 . output i32 .
do
  return sub a b .
end
```

Words in front of the type say ownership, mutability and presence. The ones you meet often are these.

Written	Meaning
`input xs slice u8 .`	Borrows someone's bytes for reading
`input xs mut slice u8 .`	Borrows someone's bytes for writing (must be a `proc`)
`input p ref point .` · `mut_ref point`	Borrows one value for reading · writing (chapter 12)
`input o option u64 .`	A value that may or may not be there (chapter 11)
`input out cap io .`	A capability (chapter 16)
`input comptime t type .`	A type fixed at translation time (chapter 22)

Table 5.1 — Words attached to parameter types

There is one return value. To return several, group them in a `struct` (chapter 10) or write into a `mut` slice the caller passed.

5.6 Only `neg` is unary

The arithmetic ops `add`, `sub`, `mul`, `div` and `mod` take two arguments. The only arithmetic op that takes one is `neg`, which flips the sign.

examples/ch05/neg.low

```
module negs .
rem run: flip 5
rem run: dist -3 4

fn flip input a i64 . output i64 .
do
  return neg a .
end

fn dist input a i64 . input b i64 . output i64 .
do
  return abs (sub a b) .
end
```

Output

```
$ lowentc --run flip neg.low 5
flip(5) = -5
$ lowentc --run dist neg.low -3 4
dist(-3, 4) = 7
```

`neg` is for signed types. On an unsigned value it overflows. `abs` is the absolute value, and it stops when given a value that cannot be represented as a positive, like the smallest `i64`. There are no unary operators inside the `expr` island either; to flip a sign there, write `expr 0 - a` or call `(neg a)` in parentheses.

5.7 Modifiers

Modifiers can go in front of an op head.

- `export` — exports the op so other modules can call it. When emitting native code it becomes a symbol callable from C. An op without it is used only inside its module, because exporting should be a choice, not a side effect.
- `extern` — the implementation is outside (in C). Instead of a body, a `link` clause names it (chapter 29).
- `unsafe` — does work the language cannot check. It pairs with `effects unsafe`.

```
unsafe extern proc c_area do
  input k cap c .
  input w i64 .
  input h i64 .
  output i64 .
  effects unsafe .
  link lw_c_area .
end
```

5.8 Common mistakes

Almost everyone hits these when writing their first ops. All of them are caught at compile time, so when you see one of these codes you can come back to this section.

> **Counter-example. Calling C-style, with parentheses and commas**
>
> If you know other languages, your fingers type `add3(1, 2, 3)` before you think.

examples/ch05/mistake_ccall.low

```
module mistake_ccall .
rem expect: E-VOCAB-REMOVED

fn add3 input a i32 . input b i32 . input c i32 . output i32 .
do
  return add (add a b) c .
end

fn use3 output i32 .
do
```

```
  rem ✘ called C-style with parentheses and commas --- a Lowent call is `name arg arg …`
  return add3(1, 2, 3) .
end
```

Output

```
$ lowentc --check mistake_ccall.low
12:16 E-VOCAB-REMOVED: `,` (R3) was removed — RFC-0103, 2026-08-27. It opened the NEXT o
perand of the same form, but nothing used it that way: every `,` in the corpus was a LIN
E CONTINUATION, and newlines no longer close a form, so continuing a line needs nothing
at all. A form is `head operand*` and ends at its closer `.` — just write the operands,
on as many lines as you like.
12:19 E-VOCAB-REMOVED: `,` (R3) was removed — RFC-0103, 2026-08-27. It opened the NEXT o
perand of the same form, but nothing used it that way: every `,` in the corpus was a LIN
E CONTINUATION, and newlines no longer close a form, so continuing a line needs nothing
at all. A form is `head operand*` and ends at its closer `.` — just write the operands,
on as many lines as you like.
```

A Lowent call is prefix notation: **the name, then the arguments separated by spaces**. The comma was once part of the language and has been removed, hence `E-VOCAB-REMOVED`. Calls avoid parentheses and commas so that everyone writes the same shape with as few symbols as possible — even on a phone keyboard. The fix: `return add3 1 2 3 .`

Counter-example. The number of arguments differs from the `input` clauses

examples/ch05/mistake_arity.low

```
module mistake_arity .
rem expect: E-IR-ARITY

fn add3 input a i32 . input b i32 . input c i32 . output i32 .
do
  return add (add a b) c .
end

fn use3 output i32 .
do
  rem ✘ three input clauses, but only two arguments
  return add3 1 2 .
end
```

Output

```
$ lowentc --check mistake_arity.low
mistake_arity.low:12:0 E-IR-ARITY: `add3` takes 3 arguments but 2 are given here — an op
 call is `<name> <arg>…` with exactly as many arguments as its `input` clauses
```

One `input` clause is one argument. There are no default arguments and no ops that take a varying number of arguments — the head should fix a single calling shape so that readers are never left guessing. The diagnostic tells you how many were expected and how many were given.

Counter-example. Forgetting `return` on one branch

examples/ch05/mistake_partial.low

```
module mistake_partial .
rem expect: E-RETURN-PARTIAL

fn sign input a i32 . output i32 .
do
  if gt a 0 . do
    return 1 .
  end
  rem ✘ nothing is returned when a is 0 or less --- one path falls off the end
end
```

Output

```
$ lowentc --check mistake_partial.low
mistake_partial.low:4:0 E-RETURN-PARTIAL: this op says it OUTPUTS a value, but some path
 through its body reaches the end without a `return`. Until now the tool quietly returne
d 0 there — a value that appears NOWHERE in your source (RFC-0019 G-TOTAL). Give every p
ath a `return`, or say `output void` if it really produces nothing. An exhaustive `match
` whose every arm returns counts as returning
```

An op that declares an `output` must return a value on **every path**. The old tool silently returned 0 on the missing path, and that 0 appeared nowhere in the source, so it hid bugs. It is now rejected with `E-RETURN-PARTIAL`. The fix is to return something for the remaining case after the `if` — for example, put `return 0 .` before `end`.

Counter-example. Naming an op after a builtin

examples/ch05/mistake_builtin.low

```
module mistake_builtin .
rem expect: E-NAME-BUILTIN

rem ✘ `min` is already a builtin op name --- this declaration could never be called
fn min input a u64 . input b u64 . output u64 .
do
  return add a b .
end
```

Output

```
$ lowentc --check mistake_builtin.low
mistake_builtin.low:5:0 E-NAME-BUILTIN: this name is a BUILTIN — the resolver always pic
ks the builtin, so your declaration can never be called: it exists and does not exist. T
he namespace is FLAT (no shadowing). Rename it
```

Names live in one flat space, so the same name cannot mean two things. A declaration named after a builtin op (`add`, `min`, `len`, `ok` …) could never be called, and `E-NAME-BUILTIN` says so. Local names follow the same rule. Appendix A lists the builtin names.

> **A common misconception. Calling a `fn` does something even if you ignore the result**
>
> A `fn` is pure, so throwing its result away is **the same as not calling it at all**. The compiler is free to delete such a call.
>
> **examples/ch05/discard.low**
>
> ```
> module discard .
> rem run: caller
>
> fn twice input a u64 . output u64 .
> do
>   return mul a 2 .
> end
>
> fn caller output u64 .
> do
>   rem a fn call whose result is not kept --- being pure, it leaves no trace
>   twice 5 .
>   rem to use the result, bind it to a name
>   let t u64 be twice 5 .
>   return t .
> end
> ```
>
> **Output**
>
> ```
> $ lowentc --run caller discard.low
> caller() = 10
> ```
>
> The first `twice 5 .` computes a value and leaves no trace. The answer is `10` only because the second line binds the result to `t`. If a call is meant to **do** something (print, write), it has to be a `proc`, and then its effects are written in its head.

5.9 This chapter's syntax at a glance

Shape	Meaning	Why
`fn f input a T . output R . do … end`	a pure op	the head alone tells whether results may be cached, re-ordered or dropped
`proc f … effects E . do … end`	an op that may have effects	what it does (`E`) is visible in the head
`input x T .`	one parameter	one per clause, so each name and type sits on its own line
`output T .` · `output void .`	type of the returned value · no returned value	one return value — bundle several in a `struct`
`return e .`	return a value and finish	required on every path (`E-RETURN-PARTIAL`)
`f a b c`	a call — name, then space-separated arguments	prefix notation without parentheses or commas

`f (g a) b`	a call used as an argument	parentheses mean “this is one value”
`requires c .`	a condition the caller must meet	blame lands on the caller, at entry (chapter 14)
`effects state .`	writes to the caller’s storage	the head of an op that writes through `mut` parameters
`neg a`	flip the sign (the only unary arithmetic)	a name, so it never gets confused with subtraction
`export` · `extern` · `unsafe`	exported · body in C · does unchecked things	what an op may do is visible before its name

Table 5.2 — Op syntax — shape · meaning · why it looks this way

Recap

An op is a `fn` or a `proc`, and the kind is always written. Purity is observational: mutation confined in the op counts as pure, but a write visible to the caller through a `mut` parameter needs a `proc`. A `fn` carries no `effects` clause, and a `proc`’s `effects` is a narrowing clause. Effects spread to the caller. One clause per parameter, one return value, and the only unary arithmetic op is `neg`.

6 Locals — `let` and `var`

In this chapter

6.1 The word says whether it changes 77
6.2 Write the type, or leave it to the value 78
6.3 No name without a value 79
6.4 How long a local lives 79
6.5 `if` yields no value 81
6.6 Common mistakes 81
6.7 This chapter’s syntax at a glance 84

What to know first

chapter 3, The surface · shadowing is forbidden
chapter 5, Ops · mutation confined inside an op counts as pure

Looking back

`running_total` in chapter 5 accumulated with `var` and was still a `fn`. Why was that allowed?

> **A.** Because purity is observational. `total` is a local that lives only inside the op, so the caller cannot see it change. Only writes the caller can see (through a `mut` parameter) count as effects. This chapter covers the two words that make locals and the span over which a local lives.

The need for this chapter, and its context

A value that changes is expensive for the reader, who has to ask again at every use “what is this value here?”. Lowent separates names that do not change from names that do **by word**, and closes the door on naming something without a value. Before using loops and branches in chapter 7, this chapter fixes the containers that hold values inside them.

By the end of this chapter

You will learn that `let` is immutable and `var` is mutable and changed only with `set`. You will pick up when to write the type and when to leave it out, and why a binding without a value is rejected (especially the `.5` trap). You will also see that a local lives only inside its block, that a live outer name cannot be reused inside, and that `if` is a statement that yields no value.

The questions this chapter answers

1. Can the elements of a slice bound with `let` be changed?
2. If `if` could be an expression, couldn’t we have fewer `var`s?

6.1 The word says whether it changes

A name that holds a value in an op body is a **local**. There are only two kinds.

- `let <name> <type> be <expr> .` — immutable. Once set, it does not change.
- `var <name> <type> be <expr> .` — mutable. It is changed with `set <name> <expr> .`.

examples/ch06/sumto.low

```
module sumto .
rem run: sum_to 10
rem run: sum_to 100

fn sum_to input n u64 . output u64 .
do
  var total u64 be 0 .
  var i u64 be 1 .
  while le i n . do
    set total (add total i) .
    set i (add i 1) .
  end
  return total .
end
```

Output

```
$ lowentc --run sum_to sumto.low 10
sum_to(10) = 55
$ lowentc --run sum_to sumto.low 100
sum_to(100) = 5050
```

`total` and `i` change as the loop runs, so they are `var`s. Only values that change are made with `var`; the rest are `let` by default. Using `set` on a `let` is rejected.

examples/ch06/immutable.low

```
module immutable .
rem expect: E-IMMUTABLE

fn f output u64 .
do
  let a u64 be 1 .
  set a 2 .
  return a .
end
```

Output

```
$ lowentc --check immutable.low
immutable.low:7:0 E-IMMUTABLE: this binding was made with `let`, which is IMMUTABLE — `s
et` cannot reassign it. Until now `let` and `var` meant exactly the same thing (two spel
lings, one meaning — and SPEC-002 §2.5 forbids synonyms). Use `var` if you mean to reass
ign it
```

According to the long explanation in the diagnostic, `let` and `var` once meant the same thing. Two spellings for one meaning means one of them says nothing. Today `let` **promises** immutability and the compiler checks the promise.

> **Q. Can the elements of a slice bound with `let` be changed?**
>
> **A.** What `let` forbids is **putting a different value into the name**. Whether the elements of a slice can be written is decided by the slice's type — `mut slice` allows writing elements and `slice` only reading (chapter 9). Immutability of the name and immutability of the bytes it points to are different questions.

6.2 Write the type, or leave it to the value

A local's type may be written or left out. When left out, the value decides the type.

examples/ch06/infer.low

```
module infer .
rem run: next_byte 41
rem trap: next_byte 255

fn next_byte input a u8 . output u8 .
do
  let b be add a 1 .
  return b .
end
```

Output

```
$ lowentc --run next_byte infer.low 41
next_byte(41) = 42
$ lowentc --run next_byte infer.low 255
== ir diagnostics (1) ==
```

```
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
```

In `let b be add a 1 .`, `a` is a `u8`, so `b` is a `u8` too. That is why adding 1 to 255 overflows `u8` and stops. If the type cannot be determined from the value, it is rejected and must be written.

This book usually writes the type. The width is the boundary of overflow (chapter 4), so when the type is visible in the source, the places that can stop are visible with it. Types are left out only for short intermediate values or where the type is obvious.

6.3 No name without a value

There must be a value after `be`. There is no way to make a name first and give it a value later.

examples/ch06/novalue.low

```
module novalue .
rem expect: E-LET-NOVALUE

fn half output f64 .
do
  let x f64 be .5 .
  return x .
end
```

Output

```
$ lowentc --check novalue.low
novalue.low:6:0 E-LET-NOVALUE: this binding has NO VALUE — `be` is followed by nothing,
and the tool used to quietly bind 0 there: a value that appears NOWHERE in your source.
If you wrote a float like `.5`, that is the cause: a LEADING DOT is a form CLOSER here,
not part of a number, so the value vanished before it was ever read. Write `0.5`. Otherw
ise give the binding a value — this language has no way to declare a name and fill it in
 later
```

This is not a blank left by mistake. The author believes they wrote the value `.5`. But `.5` is not a floating-point literal — the free-standing full stop closes the form, and nothing is left after `be`. The old tool quietly put 0 there, and a value that appears nowhere in the source got into the program. Now it is rejected, and the diagnostic names the trap. A floating-point half is written `0.5`.

> **A common misconception. An uninitialised variable is zero**
>
> In C, an uninitialised local has an unspecified value. Some languages fill in zero. Lowent has neither — there is no such thing as an uninitialised variable. There is no moment at which a value is undecided, so there is no defect that reads at that moment.

6.4 How long a local lives

A local lives until the end of the block it was made in. When the block ends, the name can be used again.

examples/ch06/scope.low

```
module scope .
rem run: pick 5
rem run: pick 0

fn pick input a u64 . output u64 .
do
  var result u64 be 0 .
  if gt a 1 . do
    let doubled u64 be mul a 2 .
    set result doubled .
  end
  if le a 1 . do
    let doubled u64 be 1 .
    set result doubled .
  end
  return result .
end
```

Output

```
$ lowentc --run pick scope.low 5
pick(5) = 10
$ lowentc --run pick scope.low 0
pick(0) = 1
```

The two `if` blocks each make a `doubled`. When the first block ends, the first `doubled` is gone, so the second block may make one with the same name. There is no moment at which both names are **alive at once**.

On the other hand, an inner block making a name that is still alive outside is rejected.

examples/ch06/outer.low

```
module outer .
rem expect: E-NAME-SHADOW

fn f input a u64 . output u64 .
do
  let t u64 be 1 .
  if gt a 1 . do
    let t u64 be 2 .
    return t .
  end
  return t .
end
```

Output

```
$ lowentc --check outer.low
outer.low:8:0 E-NAME-SHADOW: an inner block binds a name that is already bound OUTSIDE i
t. The namespace is FLAT (SPEC-002 2.10: no shadowing) — the outer value is still alive
and the same letters now point at a different one. Rename the inner binding
outer.low:11:0 E-NAME-SCOPE: this name was declared INSIDE a block and is read outside i
t. A block is where a name lives (§6.5.1): on the path that did not enter the block the
```

```
name never existed, and the tool used to answer 0 there — a value that appears nowhere i
n the source. Declare it before the block (`var … be 0 .`) and set it inside
```

Inside the inner block `t` would be 2, and outside it 1 again. When the same letters mean different values on different lines, the reader thinks of the wrong value. It is the same rule as the parameter shadowing in chapter 3 — the name space is flat.

6.5 `if` yields no value

`if` is a statement. It cannot stand where an expression is expected.

examples/ch06/ifvalue.low

```
module ifvalue .
rem expect: E-IF-VALUE

fn pick input a u64 . output u64 .
do
  let x u64 be if gt a 1 . 5 else 6 .
  return x .
end
```

Output

```
$ lowentc --check ifvalue.low
ifvalue.low:6:0 E-IF-VALUE: `if` is a STATEMENT — it gives no value (§6.5.2 (4)). Choose
 the value in each branch instead: `var x u64 be 6 .  if c do set x 5 . end`, or `return
` from each branch
```

To decide a value per branch, do what the diagnostic suggests: make a `var` with a default and `set` it inside the branches, or `return` from each branch. When the value is one of several cases over and over, `match` (chapter 7) is the better tool.

> **Q. If `if` could be an expression, couldn't we have fewer `var`s?**
>
> **A.** We could. In exchange, blocks would enter expressions, and on meeting a `return` or `break` inside such a block you would have to work out where that expression leaves to. Lowent chose to separate expressions from statements. Expressions make values; statements change flow.

6.6 Common mistakes

> **Counter-example. Misspelling the name in a `set`**
>
> **examples/ch06/mistake_typo.low**
>
> ```
> module mistake_typo .
> rem expect: E-IR-UNDEF
>
> fn bump output u64 .
> do
>   var total u64 be 0 .
> ```

```
  rem ✘ a misspelt name --- `totl` was never declared
  set totl (add total 1) .
  return total .
end
```

Output

```
$ lowentc --check mistake_typo.low
mistake_typo.low:8:0 E-IR-UNDEF: set of an undeclared name
```

`set` only puts a new value into a `var` that **already exists**. Some languages quietly create a new variable when you assign to an unknown name; here it is rejected with `E-IR-UNDEF`. If a typo became a new variable, the real one would never change and the program would silently compute the wrong answer. That is why declaring (`var`) and changing (`set`) are different words.

Counter-example. Assigning with `=`

examples/ch06/mistake_equals.low

```
module mistake_equals .
rem expect: E-CHAR

fn bump output u64 .
do
  var total u64 be 0 .
  rem ✘ `=` is not a symbol of this language --- putting a new value in is `set`
  total = add total 1 .
  return total .
end
```

Output

```
$ lowentc --check mistake_equals.low
8:9 E-CHAR: unexpected character
8:9 E-FORM-UNEXPECTED: unexpected token in form
```

In most languages `=` means assignment, while in mathematics it means equality. One symbol switching between two meanings has produced bugs like `if (x = 0)`. Lowent uses no symbol for either: declaring is `let`/`var … be`, changing is `set`, and asking whether two values are equal is `eq`. `=` is not a character the language knows at all, hence `E-CHAR`.

Counter-example. Naming a local after a common English word

examples/ch06/mistake_localname.low

```
module mistake_localname .
rem expect: E-NAME-BUILTIN

fn size input xs slice u8 . output u64 .
do
  rem ✘ `count` is the name of a builtin op --- it cannot be a local name
  let count u64 be len xs .
```

```
  return count .
end
```

Output

```
$ lowentc --check mistake_localname.low
mistake_localname.low:7:0 E-NAME-BUILTIN: a local takes the name of a BUILTIN op. The na
mespace is FLAT (no shadowing), so this name now means two things — and which one it mea
ns decides how the SENTENCE IS BRACKETED: `f len data` reads as `f(len(data), …)` if `le
n` is the builtin (arity 1) and as `f(len, data)` if it is your local (arity 0). Same le
tters, different tree, no error. That is RFC-0046 P1. Rename the local
```

Names such as `count`, `len`, `min`, `max`, `ok` and `avg` are already builtin ops. The name space is flat, so if a local took one of them, `f count data` would group differently depending on whether `count` is the builtin call or the local. The same letters would build a different tree, so `E-NAME-BUILTIN` rules it out. Use names like `n`, `total` or `size`.

Counter-example. Reading a local outside the block that declared it

examples/ch06/mistake_outside.low

```
module mistake_outside .
rem expect: E-NAME-SCOPE

fn big_or_zero input a u64 . output u64 .
do
  if gt a 3 . do
    let big u64 be mul a 10 .
  end
  rem ✘ `big` only lives inside the block above --- it is read outside
  return big .
end
```

Output

```
$ lowentc --check mistake_outside.low
mistake_outside.low:10:0 E-NAME-SCOPE: this name was declared INSIDE a block and is read
 outside it. A block is where a name lives (§6.5.1): on the path that did not enter the
block the name never existed, and the tool used to answer 0 there — a value that appears
 nowhere in the source. Declare it before the block (`var … be 0 .`) and set it inside
```

`big` was declared inside the `if` block, so it disappears when the block ends. Reading that name outside the block is rejected with `E-NAME-SCOPE`; it used to **quietly give 0** on the path that never entered the block — a value that appears nowhere in the source. Declare a value you need after the block **before** the block.

examples/ch06/outside_fixed.low

```
module outside_fixed .
rem run: big_or_zero 5
rem run: big_or_zero 1

fn big_or_zero input a u64 . output u64 .
```

```
do
  rem a value used after the block is declared before it; the block only sets it
  var big u64 be 0 .
  if gt a 3 . do
    set big (mul a 10) .
  end
  return big .
end
```

Output

```
$ lowentc --run big_or_zero outside_fixed.low 5
big_or_zero(5) = 50
$ lowentc --run big_or_zero outside_fixed.low 1
big_or_zero(1) = 0
```

Now the 0 is a default written in the source. The answer is the same, but **where the 0 comes from** is visible.

6.7 This chapter's syntax at a glance

Shape	Meaning	Why
`let x T be e .`	a name that never changes	immutable by default — only changing values stand out
`var x T be e .`	a name that may change	say that it will change at the moment you declare it
`set x e .`	put a new value into a `var`	declaring and changing are different words — a typo never becomes a new variable
`let x be e .`	let the value decide the type	only for short intermediate values — the width is the overflow boundary
the value after `be`	always required	there is no such thing as an uninitialised variable
the end of a block	locals declared inside it disappear	names live close to where they are used
re-declaring an outer name inside	rejected (`E-NAME-SHADOW`)	the same letters point to one value only
`eq a b`	ask whether two values are equal	there is no `=` — assignment and equality never mix

Table 6.1 — Local syntax — shape · meaning · why it looks this way

Recap

There are two kinds of local, `let` (immutable) and `var` (mutable), and `set` works only on a `var`. The type may be written or left to the value, but a value after `be` is required. A local lives until the end of its block, and a live name cannot be made again even in an inner block. `if` is a statement that yields no value.

7 Flow — branches, loops and leaving early

In this chapter

7.1 Conditions and loops 86
7.2 `guard` — turning a condition into a fact 87
7.3 Every path returns a value 89
7.4 `match` — every case, none missing 90
7.5 `panic` is an effect 91
7.6 Common mistakes 92
7.7 This chapter's syntax at a glance 96

What to know first

chapter 4, Numbers · booleans are not numbers
chapter 6, Locals · `if` is a statement that yields no value

Looking back

In chapter 6, why was `let x u64 be if gt a 1 . 5 else 6 .` rejected, and how do you set a value per branch?

> **A.** `if` is a statement that yields no value, so it cannot stand in an expression position (`E-IF-VALUE`). To set a value per branch, make a `var` with a default and `set` it in the branches, or `return` from each branch. This chapter covers the branches themselves, loops, and the rules for leaving early.

The need for this chapter, and its context

Flow statements exist in every language, so there seems to be nothing new to learn. But Lowent's flow statements come with **checks**. Conditions must be booleans, the `else` of a `guard` must leave, an op that returns a value must return on every path, and a `match` must cover every case. These checks remove "on this path there is no value" defects at translation time. The promises of flow are set up before data (Part III).

By the end of this chapter

You will learn to use `if … else`, `while`, `for`, `break` and `continue`. You will see how `guard` turns a condition into **a fact about the code below it**, and why an `else` that does not leave is rejected. You will also meet the rule that every path must return a value, `match` over numbers and ranges with its exhaustiveness check, and the fact that `panic` is an effect.

The questions this chapter answers

1. How do you write a loop counting from 0 to `n` with `for`?
2. Besides exhaustiveness, what makes `match` better than an `if` chain?

7.1 Conditions and loops

`if` runs a block when its condition is true, and `else` gives the block for false. `while` repeats while its condition is true, and `for` walks the elements of a slice in order. Every condition must be a `bool`.

examples/ch07/loops.low

```
module loops .
rem run: count_big 10
rem run: first_zero [5,3,0,9]
rem run: first_zero [5,3]
rem run: odd_sum [1,2,3,4,5]

fn count_big input n u32 . output u32 .
  requires le n 100 .
do
  var total u32 be 0 .
  var i u32 be 0 .
  while lt i n . do
    if gt i 5 . do
      set total (add total 1) .
    end
    set i (add i 1) .
  end
  return total .
end

fn first_zero input xs slice u8 . output u64 .
do
  var i u64 be 0 .
  while lt i (len xs) . do
    if eq (index xs i) 0 . do
      break .
    end
    set i (add i 1) .
  end
  return i .
end

fn odd_sum input xs slice u8 . output u64 .
do
```

```
  var acc u64 be 0 .
  for x xs do
    if eq (mod x 2) 0 . do
      continue .
    end
    set acc (add acc (widen u64 x)) .
  end
  return acc .
end
```

Output

```
$ lowentc --run count_big loops.low 10
count_big(10) = 4
$ lowentc --run first_zero loops.low [5,3,0,9]
first_zero([5,3,0,9]) = 2
  arg0 (written) = [5,3,0,9]
$ lowentc --run first_zero loops.low [5,3]
first_zero([5,3]) = 2
  arg0 (written) = [5,3]
$ lowentc --run odd_sum loops.low [1,2,3,4,5]
odd_sum([1,2,3,4,5]) = 9
  arg0 (written) = [1,2,3,4,5]
```

- `count_big` counts from 0 up to `n` and counts only numbers greater than 5. Look at the shape `while lt i n . do … end` — the condition is a form too, so it is closed with a full stop, and `do` opens the body after it.
- `first_zero` leaves the loop with `break` when it meets a 0, and returns the length if there is none.
- `odd_sum` walks the elements with `for x xs do … end` and skips even ones with `continue`. The `for` name `x` has the slice's element type (`u8`) and lives only inside the block.

Writing `for x in xs` is rejected with `E-VOCAB-REMOVED`. The thing to walk comes right after the name.

> **Q. How do you write a loop counting from 0 to `n` with `for`?**
>
> **A.** `for` is a tool for walking slices and does not take a numeric range directly. Counting loops are written with `while` and `var`. Having one shape for counting loops makes the range of the counter easy for the compiler to see and keeps the analysis that removes bounds checks simple. Filtering, counting and collecting over a slice are `pipe`'s job (chapter 24).

7.2 `guard` — turning a condition into a fact

`guard <condition> . else <leave> .` **leaves on the spot** if the condition is not true. The leaving statements are `return`, `break`, `continue` and `panic`.

examples/ch07/guards.low

```
module guards .
rem run: head_or_zero [7,8]
rem run: head_or_zero []
rem run: grade 95
```

```
rem run: grade 42

fn head_or_zero input data slice u8 . output u8 .
do
  guard ge (len data) 1 . else return 0 .
  return index data 0 .
end

fn grade input score u8 . output u8 .
do
  guard le score 100 . else return 0 .
  if ge score 90 . do
    return 65 .
  end else do
    if ge score 60 . do
      return 66 .
    end
  end
  return 70 .
end
```

Output

```
$ lowentc --run head_or_zero guards.low [7,8]
head_or_zero([7,8]) = 7
  arg0 (written) = [7,8]
$ lowentc --run head_or_zero guards.low []
head_or_zero([]) = 0
  arg0 (written) = []
$ lowentc --run grade guards.low 95
grade(95) = 65
$ lowentc --run grade guards.low 42
grade(42) = 70
```

The code below the `guard` in `head_or_zero` lives only in a world where the slice is not empty. That is why `index data 0` is safe. `grade` filters out out-of-range scores first with `guard`, and splits the rest with `if … end else do … end` below it.

`guard` is not another name for `if not`, because its `else` must leave. If it does not, the code is rejected.

examples/ch07/fallthrough.low

```
module fallthrough .
rem expect: E-GUARD-FALLTHROUGH

fn clamp5 input n u32 . output u32 .
do
  guard le n 5 . else do
    let capped u32 be 5 .
  end
  return n .
end
```

Output

```
$ lowentc --check fallthrough.low
fallthrough.low:6:0 E-GUARD-FALLTHROUGH: this `guard`'s `else` FALLS THROUGH — it must L
EAVE (`return` / `break` / `continue` / `panic`). If it falls through, the code AFTER th
e guard wrongly believes the condition holds. Leaving is the WHOLE point of `guard`: it
is what lets the rest of the op assume the condition. (SPEC-002 D6 always said so and no
thing enforced it — which made `guard` an exact synonym of `if not`.) If you do not mean
 to leave, write `if`
```

If the `else` flows on instead of leaving, the code below the `guard` believes `n` is at most 5 while it is not. Rather than letting code believe an unreliable fact, translation is refused. If you do not mean to leave, use `if`.

> **A common misconception. `guard` is syntactic sugar for shorter code**
>
> Being shorter is a side effect. The real point of `guard` is **handing a fact to the compiler and the reader**. After a `guard` the compiler knows the condition is true and uses it to remove bounds checks. You can write the same meaning with `if … do return … end` and get the same behaviour, but leaving is not guaranteed by the grammar.

7.3 Every path returns a value

An op that returns a value must return one on **every path**.

examples/ch07/partial.low

```
module partial .
rem expect: E-RETURN-PARTIAL

fn sign_flag input a i32 . output u8 .
do
  if gt a 0 . do
    return 1 .
  end
end
```

Output

```
$ lowentc --check partial.low
partial.low:4:0 E-RETURN-PARTIAL: this op says it OUTPUTS a value, but some path through
 its body reaches the end without a `return`. Until now the tool quietly returned 0 ther
e — a value that appears NOWHERE in your source (RFC-0019 G-TOTAL). Give every path a `r
eturn`, or say `output void` if it really produces nothing. An exhaustive `match` whose
every arm returns counts as returning
```

On the path where `a` is 0 or less, there is no `return`. The old tool quietly returned 0 on that path — a value that appears nowhere in the source. Now it is rejected. The statements recognised as returning are `return`, an `if … else` where both branches return, and a `match` where every arm returns. `while`, `for` and `guard` have ways out, so they do not count as returning by themselves. That is why a loop is always followed by a `return`.

An op that returns nothing (`output void`) may run to the end of its body without `return`. Where it ends is where it returns.

7.4 `match` — every case, none missing

`match` splits a value into cases. Unlike an `if` chain, it **must handle every case**.

examples/ch07/bands.low

```
module bands .
rem run: band 9
rem run: band 10
rem run: band 200
rem run: half 128
rem run: big 101

fn band input x u8 . output u8 .
do
  match x do
    case 0 to 9 . do return 1 . end
    case 10 to 19 . do return 2 . end
    case _ . do return 0 . end
  end
end

fn half input b u8 . output u8 .
do
  match b do
    case 0 to 127 . do return 0 . end
    case 128 to 255 . do return 1 . end
  end
end

fn big input x u8 . output u8 .
do
  match x do
    case y when gt y 100 . do return 1 . end
    case _ . do return 0 . end
  end
end
```

Output

```
$ lowentc --run band bands.low 9
band(9) = 1
$ lowentc --run band bands.low 10
band(10) = 2
$ lowentc --run band bands.low 200
band(200) = 0
$ lowentc --run half bands.low 128
half(128) = 1
$ lowentc --run big bands.low 101
big(101) = 1
```

- `band` splits by `lo to hi` ranges (inclusive at both ends). Integers have many cases, so a `case _ .` covering the rest is required.
- `half` needs no _ because its two ranges cover `u8`'s 0 … 255 with no gap.
- `case y when gt y 100 .` in `big` binds the whole value to `y` and narrows it further with a guard. An arm with a guard might be false, so it does not count towards exhaustiveness; _ is still needed.

A gap is rejected.

examples/ch07/inexhaustive.low

```
module inexhaustive .
rem expect: E-MATCH-INEXHAUSTIVE

fn band input x u8 . output u8 .
do
  match x do
    case 0 to 9 . do return 1 . end
    case 11 to 255 . do return 2 . end
  end
end
```

Output

```
$ lowentc --check inexhaustive.low
inexhaustive.low:6:0 E-MATCH-INEXHAUSTIVE: this `match` on an integer is not exhaustive
— the literals/ranges leave a gap, so a `case _ .` wildcard (or `else`) is required, OR
the ranges must TILE the whole domain of the scrutinee's type (e.g. `0 to 127` + `128 to
 255` on a u8) (RFC-0020 §6.4; MM4 tiling)
```

10 falls in no arm. Writing the same case twice, or an arm after _, is rejected with `E-MATCH-REDUNDANT`. It is an error, not a warning — an arm that can never run usually hides a defect.

> **Q. Besides exhaustiveness, what makes `match` better than an `if` chain?**
>
> **A.** It shows when cases grow. Add a variant to an enum and every `match` over that enum stops at translation and tells you where to fix it (chapter 10). An `if` chain lets the new case slide silently into its last `else`. Also, `--ir` has the tool report the dispatch cost, such as the number of arms and the worst-case number of comparisons.

7.5 `panic` is an effect

`panic` stops the program immediately. It is an irreversible stop, so it is an effect, and an op that uses it must be a `proc` declaring `effects panic`.

examples/ch07/panics.low

```
module panics .
rem run: checked 3
rem trap: checked 0

proc checked input n u32 . output u32 . effects panic .
do
  if eq n 0 . do
    panic "n must not be zero" .
  end
  return div 100 n .
end
```

Output

```
$ lowentc --run checked panics.low 3
checked(3) = 33
$ lowentc --run checked panics.low 0
== ir diagnostics (1) ==
0:0 E-VM-PANIC: the program called `panic` — this is an unrecoverable stop, and it is NO
T a contract violation (the code chose to stop, it did not break a promise)
```

The VM reports `E-VM-PANIC` and makes clear that **this is not a contract violation** — the code chose to stop; it did not break a promise. A pure `fn` cannot `panic`. But stops the processor raises for overflow or contract violations still happen inside a `fn`. Those are not something the op did but the processor making promises hold.

Use `panic` only for situations that cannot be recovered from. Failures a caller can handle are returned as values (chapter 11). How to tell the two apart is covered in chapter 17.

7.6 Common mistakes

Counter-example. Writing `for x in xs`

examples/ch07/mistake_forin.low

```
module mistake_forin .
rem expect: E-VOCAB-REMOVED

fn total input xs slice u8 . output u64 .
do
  var t u64 be 0 .
  rem ✘ wrote `in` --- `for` is `for <name> <slice> do`
  for x in xs do
    set t (add t (widen u64 x)) .
  end
  return t .
end
```

Output

```
$ lowentc --check mistake_forin.low
8:0 E-VOCAB-REMOVED: `in` is gone. In a loop write `for <name> <slice> do` — the slice f
ollows the name and `do` marks the body, so `in` carried nothing. As access, `b in a` wa
s a reverse spelling of `field a b`: write `field a b` / `index a i` (the glued dot `a.b
` is refused too, `E-FIELD-GLUED`)
```

`for` is `for <name> <slice> do`. The thing to walk comes right after the name, and `do` already marks where the body starts, so `in` would carry nothing. It was removed so that one meaning has one spelling. The fix: `for x xs do`.

Counter-example. Chaining branches with `elif`

examples/ch07/mistake_elif.low

```
module mistake_elif .
rem expect: E-BLOCK-NOHEAD
```

```
fn grade input n u64 . output u64 .
do
  if lt n 50 . do
    return 0 .
  rem ✘ `elif` is not a word --- a following branch is `end else if … do`
  end elif lt n 80 . do
    return 1 .
  end
  return 2 .
end
```

Output

```
$ lowentc --check mistake_elif.low
9:22 E-BLOCK-NOHEAD: a `do … end` block needs a head that owns it — `if … do`, `while …
do`, `fn … do`, `make T do`, `region … do` … A bare block is not in the grammar; it used
 to pass `--check` and then could not be lowered. Put its statements where they belong,
or give it its head
```

`elif`, `elsif` and `else if`: vary between languages. Lowent joins words it already has: close the previous block with `end` and add `else if`. `elif` is not a word, so `elif lt n 80 .` is read as a statement of its own, and the `do … end` after it becomes a block with no head to open it (`E-BLOCK-NOHEAD`) — a `do … end` is always opened by a head such as `if`, `while` or `fn`.

examples/ch07/elseif.low

```
module elseif .
rem run: grade 30
rem run: grade 70
rem run: grade 95

fn grade input n u64 . output u64 .
do
  if lt n 50 . do
    return 0 .
  rem to chain branches, close the previous block with `end` and add `else if`
  end else if lt n 80 . do
    return 1 .
  end else do
    return 2 .
  end
end
```

Output

```
$ lowentc --run grade elseif.low 30
grade(30) = 0
$ lowentc --run grade elseif.low 70
grade(70) = 1
$ lowentc --run grade elseif.low 95
grade(95) = 2
```

The last branch is `end else do … end`. When there are three or more branches that all split one value, `match` is a better fit.

Counter-example. Putting `else` inside the block, C style

examples/ch07/mistake_innerelse.low

```
module mistake_innerelse .
rem expect: E-STMT-ELSE

fn pick input a u8 . output u8 .
do
  var r u8 be 0 .
  if gt a 5 . do
    set r 1 .
  rem ✘ `else` sits on its own line inside the block, C style --- the previous block mus
t be closed with `end else do`
  else
    set r 2 .
  end
  return r .
end
```

Output

```
$ lowentc --check mistake_innerelse.low
mistake_innerelse.low:10:0 E-STMT-ELSE: `else` sits INSIDE the block, the way C writes i
t. Here a block is closed before the other arm opens: `if <cond> . do … end else do … en
d`. Written this way the arm used to be accepted by every static check and then dropped
at lowering — the VM stopped with an unsupported body and the native build silently left
 the op out
```

In C and several other languages `else` follows the previous block, as in `if … { … } else { … }`. In Lowent `else` comes **after** the previous block is closed with `end` (`end else do`). With `else` on its own line inside the block the program is rejected with `E-STMT-ELSE`; it used to translate and then stop at run time with `E-VM-UNSUP`, while the native build left the op out entirely.

Counter-example. Using a symbol such as `<` in a condition

examples/ch07/mistake_less.low

```
module mistake_less .
rem expect: E-CHAR

fn upto input n u64 . output u64 .
do
  var i u64 be 0 .
  rem ✘ `<` is not a symbol of this language --- the comparison is `lt i n`
  while i < n . do
    set i (add i 1) .
  end
  return i .
end
```

Output

```
$ lowentc --check mistake_less.low
8:11 E-CHAR: unexpected character
8:11 E-FORM-UNEXPECTED: unexpected token in form
```

Comparisons are words — `lt` (less than), `le` (less or equal), `gt`, `ge`, `eq`, `ne`. `<` is a character the language does not know, hence `E-CHAR`. The choice spares you from memorising symbol precedence, and long arithmetic has the `expr` island (chapter 8). The fix: `while lt i n . do`.

Counter-example. `continue` skipping the increment

If you count with `while` and `continue` in the middle of the body, you also skip the `set i (add i 1) .` placed below it.

```
fn odd_count input xs slice u8 . output u64 .
do
  var n u64 be 0 .
  var i u64 be 0 .
  while lt i (len xs) . do
    if eq (mod (index xs i) 2) 0 . do
      continue .
    end
    set n (add n 1) .
    set i (add i 1) .
  end
  return n .
end
```

At the first even number, `i` stops increasing and the loop never ends. Neither compilation nor run-time checks catch this — running forever is not an overflow. Move the increment to the **top** of the body (keeping the pre-increment value for indexing), or, if you are walking elements, use `for x xs do` in the first place. With `for`, moving to the next element is the language's job, so this bug has nowhere to live.

There are two ways to write the slot that takes the rest. `case _ .` and a final `else` do the same thing.

examples/ch07/matchelse.low

```
module matchelse .
rem run: band 1
rem run: band 3
rem run: band 99

fn band input n u64 . output u64 . do
  match n do
    case 1 do
      return 10 .
    end
    case 2 to 5 do
      return 20 .
    end
    rem catch the rest with `else` --- the same place as `case _ .`
    else do
      return 99 .
    end
```

```
  end
end
```

Output

```
$ lowentc --run band matchelse.low 1
band(1) = 10
$ lowentc --run band matchelse.low 3
band(3) = 20
$ lowentc --run band matchelse.low 99
band(99) = 99
```

Nothing may follow an `else` — it has already taken everything, so a later arm never runs and is refused with `E-MATCH-REDUNDANT`.

> **A common misconception. `match` arms fall through like C's `switch`**

examples/ch07/nofall.low

```
module nofall .
rem run: label 1
rem run: label 2
rem run: label 9

fn label input c u64 . output u64 .
do
  var score u64 be 0 .
  match c do
    rem only the matching arm runs --- it never falls into the next arm, so no `break` i
s needed
    case 1 . set score (add score 10) .
    case 2 . set score (add score 20) .
    case _ . set score (add score 1) .
  end
  return score .
end
```

Output

```
$ lowentc --run label nofall.low 1
label(1) = 10
$ lowentc --run label nofall.low 2
label(2) = 20
$ lowentc --run label nofall.low 9
label(9) = 1
```

Only **the one** matching arm runs, and it never falls into the next. There is no `break` to remember and no bug from forgetting it. To do the same thing for two cases, join the arms with `or` (chapter 11).

7.7 This chapter's syntax at a glance

Shape	Meaning	Why

`if c . do … end`	run the block when the condition is true	the condition is a form closed by a stop; the body opens with `do`
`if c . do … end else do … end`	one of two	openers and closers always pair up
`… end else if c2 . do … end`	chaining branches	joins existing words instead of adding `elif`
`while c . do … end`	repeat while the condition is true	the one shape for counting loops
`for x xs do … end`	each element of a slice in turn	moving to the next element is the language's job
`break . · continue .`	leave the loop · go to the next round	statements that change the flow
`guard c . else return … .`	leave unless the condition holds — afterwards it is a fact	`else` must always leave
`return e .`	return a value and finish	on every path of an op that produces a value
`match v do case … . statement … end`	split by cases	complete and non-overlapping — no fall-through
`case 1 to 9 . · case _ . · case y when c .`	a range · everything else · a guarded arm	integers have many cases, so `_` is often needed
`panic "…" .`	an irreversible stop (an effect)	only in a `proc` that declares `effects panic`

Table 7.1 — Control-flow syntax — shape · meaning · why it looks this way

Recap

Conditions must be `bool`s. `while` repeats on a condition and `for` over a slice, and `break` and `continue` change the flow. The `else` of a `guard` must leave, and after the `guard` the condition is a fact. An op that returns a value must return on every path. `match` must cover cases with no gaps and no overlaps. `panic` is an effect.

8 Expressions — prefix notation and the `expr` island

In this chapter

8.1 Nothing to memorise in prefix notation 98
8.2 The island's precedence table 99
8.3 What the island cannot do 100
8.4 Short-circuiting guards conditions 101
8.5 Expressions computed at translation time 102
8.6 Common mistakes 103
8.7 This chapter's syntax at a glance 106

What to know first

chapter 3, The surface · the name first, then the arguments
chapter 4, Numbers · bitwise operations are written as words
chapter 7, Flow · `match` must cover every case

Looking back

Why did chapter 3 reject `expr a lt b lt c`? What did the diagnostic say to write instead?

> **A.** Chained comparisons read as "a < b and b < c" in mathematics but as `(a < b) < c` in many languages. Since the same spelling means different things to people and to the machine, it was rejected with `E-EXPR-CHAIN`. The diagnostic said to write `expr (a lt b) and (b lt c)`. This chapter covers every rule inside that island.

The need for this chapter, and its context

Expressions are the most used part of the grammar. In Lowent, prefix notation is the default and infix is allowed only inside an `expr` island. You need to understand this two-layer structure to read prefix expressions full of parentheses, and to know what the island does not allow so you do not get lost. As the last chapter of Part II, it gathers the rules of expressions seen piecemeal so far.

By the end of this chapter

You will learn why prefix notation has no precedence, and the island's short precedence table. You will pick up that calling an op inside the island needs parentheses, that there are no unary operators, and that comparisons mixed with `and` and `or` are parenthesised. You will also see how short-circuiting `and` and `or` keep conditions safe, and `size_of` and `comptime`, which are computed at translation time.

The questions this chapter answers

1 The VM shows `in_range(5, 1, 9) = 1`. Has the `bool` become a number?

8.1 Nothing to memorise in prefix notation

In prefix notation the operation's name comes first and its arguments follow. An argument that is itself a form is wrapped in parentheses. This shape has no precedence; the parentheses say everything about what is computed first.

examples/ch08/island.low

```
module island .
rem run: score 3 4
rem run: score_prefix 3 4
rem run: grouped 3 4
rem run: in_range 5 1 9
rem run: spread 3 9
```

```
fn score input a u32 . input b u32 . output u32 .
  requires le a 1000 .
  requires le b 1000 .
do
  return expr a + b * 2 .
end

fn score_prefix input a u32 . input b u32 . output u32 .
  requires le a 1000 .
  requires le b 1000 .
do
  return add a (mul b 2) .
end

fn grouped input a u32 . input b u32 . output u32 .
  requires le a 1000 .
  requires le b 1000 .
do
  return expr (a + b) * 2 .
end

fn in_range input x u32 . input lo u32 . input hi u32 . output bool .
do
  return expr (lo le x) and (x le hi) .
end

fn spread input a u32 . input b u32 . output u32 .
do
  return expr (max a b) - (min a b) .
end
```

Output

```
$ lowentc --run score island.low 3 4
score(3, 4) = 11
$ lowentc --run score_prefix island.low 3 4
score_prefix(3, 4) = 11
$ lowentc --run grouped island.low 3 4
grouped(3, 4) = 14
$ lowentc --run in_range island.low 5 1 9
in_range(5, 1, 9) = 1
$ lowentc --run spread island.low 3 9
spread(3, 9) = 6
```

`add a (mul b 2)` in `score_prefix` and `expr a + b * 2` in `score` are the same expression. The two ops give the same answer, and since the island is translated to prefix, the run-time cost is the same too. The island is **a projection of notation**, not a different operation.

The reason for words instead of symbols is the same. `^` is exponentiation in some languages and exclusive or in others. `bit_xor` means one thing wherever it is read. And words can be read aloud.

8.2 The island's precedence table

Precedence exists only inside the island, and this table is all of it.

Level	Operations	Associativity	Prefix equivalent
5	`( )` grouping	—	Only marks grouping; not a value
4	`* /`	left	`mul` · `div`
3	`+ -`	left	`add` · `sub`
2	`eq ne lt le gt ge`	cannot chain	the prefix op of the same name
1b	`and`	left	`and` (short-circuit)
1a	`or`	left	`or` (short-circuit)

Table 8.1 — Precedence inside an `expr` island (tightest at the top)

Parentheses change the order, as in `grouped`'s `expr (a + b) * 2`. Operations not in the table — the remainder `mod`, bitwise operations, `min` and `max` — are written prefix and called in parentheses even inside the island. `spread`'s `expr (max a b) - (min a b)` has that shape.

`in_range` joins two comparisons with `and`. By the table, comparisons bind tighter than `and`, so parentheses seem unnecessary, but the compiler in this edition rejects `expr lo le x and x le hi` with `E-TYPE-LOGICAL`. When mixing comparisons with `and` or `or`, parenthesise each comparison. The reader no longer needs to recall the table either.

> **Q. The VM shows `in_range(5, 1, 9) = 1`. Has the `bool` become a number?**
>
> **A.** No. The VM's result line merely **displays** a `bool` as 0 or 1. Inside a program a `bool` does not mix with numbers (chapter 4). In the C that native code is emitted as, a `bool` is also one byte holding 0 or 1.

8.3 What the island cannot do

The island is a world of infix operators, so two things are missing.

First, calling an op without parentheses. `twice a + 1` could be `twice (a + 1)` or `(twice a) + 1`.

examples/ch08/app_bad.low

```
module app_bad .
rem expect: E-EXPR-APP

fn twice input a u32 . output u32 .
  requires le a 1000 .
do
  return mul a 2 .
end

fn use input a u32 . output u32 .
  requires le a 100 .
do
  return expr twice a + 1 .
end
```

Output

```
$ lowentc --check app_bad.low
app_bad.low:13:0 E-EXPR-APP: an op call inside an `expr` island must be parenthesised —
```

```
write `(len d) ge 4`, not `len d ge 4`. The island has infix operators only: without the
 parentheses nothing says where the argument list ends, and you and the compiler would r
ead it differently
```

As the diagnostic's example shows, a call inside the island is parenthesised, as in `(twice a) + 1`.

Second, unary operators. The island has no operator without a left operand.

examples/ch08/unary_bad.low

```
module unary_bad .
rem expect: E-EXPR-UNARY

fn flip input a i32 . output i32 .
do
  return expr - a .
end
```

Output

```
$ lowentc --check unary_bad.low
unary_bad.low:6:0 E-EXPR-UNARY: an `expr` island has INFIX operators only — there is no
unary form. `expr - 5` has nothing on the left of `-`. Write the prefix op instead (`neg
 5`, `not b`, `bit_not x`), or give the left operand. The island exists to read like ari
thmetic; a unary sign would make `a - 5` and `a (- 5)` two readings of the same letters
(SPEC-002 §2.5 · RFC-0091)
```

To flip a sign, write `neg a` in prefix, or inside the island call `(neg a)` or write `expr 0 - a`. `not b` and `bit_not x` are the same.

> **A common misconception. The island is small because it is unfinished**
>
> Its size is intentional. Infix notation is only worth having **when nobody needs to look up precedence**. The operations in the table work like school arithmetic, so everyone reads them the same way. Adding bitwise or unary operators would add things to look up rather than making code easier to read. The specification states that the island will not grow.

8.4 Short-circuiting guards conditions

`and` and `or` do not compute the right-hand side when the left-hand side already decides the answer. Using that, you can put the condition that makes the right-hand side safe on the left.

examples/ch08/shortcircuit.low

```
module shortcircuit .
rem run: safe_first_is_zero []
rem run: safe_first_is_zero [0,4]
rem run: ratio_ok 0 0
rem run: ratio_ok 10 5

fn safe_first_is_zero input xs slice u8 . output bool .
do
  return and (gt (len xs) 0) (eq (index xs 0) 0) .
end
```

```
fn ratio_ok input num u32 . input den u32 . output bool .
do
  return and (ne den 0) (ge (div num den) 2) .
end
```

Output

```
$ lowentc --run safe_first_is_zero shortcircuit.low []
safe_first_is_zero([]) = 0
  arg0 (written) = []
$ lowentc --run safe_first_is_zero shortcircuit.low [0,4]
safe_first_is_zero([0,4]) = 1
  arg0 (written) = [0,4]
$ lowentc --run ratio_ok shortcircuit.low 0 0
ratio_ok(0, 0) = 0
$ lowentc --run ratio_ok shortcircuit.low 10 5
ratio_ok(10, 5) = 1
```

`safe_first_is_zero` does not compute `index xs 0` on an empty slice, because if `gt (len xs) 0` on the left is false, the answer is already false. `ratio_ok` likewise skips the division when the denominator is 0. This shape suits conditions too small for a `guard`. If the condition is a fact about the whole op, `guard` is better (chapter 7).

8.5 Expressions computed at translation time

Some expressions have values before the program runs.

examples/ch08/compt.low

```
module compt .
rem run: table_size
rem run: pick

fn table_size output u64 .
do
  return mul (size_of u32) 16 .
end

fn pick output u32 .
do
  match comptime (add 2 3) do
    case 0 . do return 100 . end
    case 5 . do return 500 . end
    case _ . do return 999 . end
  end
end
```

Output

```
$ lowentc --run table_size compt.low
table_size() = 64
$ lowentc --run pick compt.low
pick() = 500
```

`size_of u32` gives the number of bytes of one value of that type at translation time. It lets code that takes a type as a parameter compute its costs honestly instead of assuming the widest type (chapter 22).

`comptime (add 2 3)` marks an expression to be computed at translation time. When the value a `match` splits on is a translation-time constant, the `match` folds to the one matching arm and the run-time comparisons disappear. The folded-away arms are still type-checked, unlike code removed with C's `#ifdef`, which is not even checked. `config <option>`, which reads the build configuration, is also used as a translation-time constant (chapter 31).

8.6 Common mistakes

Comparison (level 2) binds tighter than `and` (level 1b), so mixing them without parentheses has exactly one meaning.

examples/ch08/mixcmp.low

```
module mixcmp .
rem run: in_range 3 1 5
rem run: in_range 9 1 5

fn in_range input x u64 . input lo u64 . input hi u64 . output bool .
do
  rem comparison binds tighter than `and`, so mixing without parentheses has one meaning
  return expr lo le x and x le hi .
end
```

Output

```
$ lowentc --run in_range mixcmp.low 3 1 5
in_range(3, 1, 5) = 1
$ lowentc --run in_range mixcmp.low 9 1 5
in_range(9, 1, 5) = 0
```

Writing the parentheses anyway spares the reader from recalling the table. Below is the same thing written that way.

examples/ch08/mixcmp_fixed.low

```
module mixcmp_fixed .
rem run: in_range 5 1 9
rem run: in_range 12 1 9

fn in_range input x u64 . input lo u64 . input hi u64 . output bool .
do
  return expr (lo le x) and (x le hi) .
end
```

Output

```
$ lowentc --run in_range mixcmp_fixed.low 5 1 9
in_range(5, 1, 9) = 1
$ lowentc --run in_range mixcmp_fixed.low 12 1 9
in_range(12, 1, 9) = 0
```

Counter-example. Writing the remainder as `%`

examples/ch08/mistake_percent.low

```
module mistake_percent .
rem expect: E-CHAR

fn last_digit input a u64 . output u64 .
do
  rem ✘ `%` is not an island operator --- call `mod` in parentheses for the remainder
  return expr a % 10 .
end
```

Output

```
$ lowentc --check mistake_percent.low
7:17 E-CHAR: unexpected character
7:17 E-FORM-UNEXPECTED: unexpected token in form
```

The island has exactly the operators in the table. `%` is the remainder in C, but languages disagree on its sign for negative numbers, so the same symbol gives different answers. Lowent's remainder is the one name `mod`, and its sign follows the divisor (chapter 4). Inside the island, call it in parentheses.

examples/ch08/percent_fixed.low

```
module percent_fixed .
rem run: last_digit_plus 1234 1

fn last_digit_plus input a u64 . input k u64 . output u64 .
do
  rem operations missing from the table are written prefix inside the island, wrapped in
 parentheses
  return expr (mod a 10) + k .
end
```

Output

```
$ lowentc --run last_digit_plus percent_fixed.low 1234 1
last_digit_plus(1234, 1) = 5
```

Counter-example. Writing powers as `^`, or using `pow` on integers

`^` is not an island operator (`E-CHAR`) — it means power in some languages and exclusive or in others, so it was left out. Square an integer with `mul a a`. And `pow` is the power of **floating-point** numbers. This edition's tool does not reject `pow` on integers; it gives a wrong value.

examples/ch08/mistake_pow.low

```
module mistake_pow .
rem expect: E-TYPE-KIND

fn square input a u64 . output u64 .
do
  rem ✘ `pow` on an integer; transcendental functions are floating-point only
```

```
  return pow a 2 .
end
```

Output

```
$ lowentc --check mistake_pow.low
mistake_pow.low:7:0 E-TYPE-KIND: this is a floating-point-only operation (canon §6.3.9)
and it was given an INTEGER. `pow` used to answer anyway — and the answer was wrong (`po
w 5 2` gave 5 on both backends), which is the worst kind of wrong: quiet. Convert first
(`cast f64 n`), or use repeated multiplication for an integer power
```

The canon makes `pow`, `sqrt`, `sin`, `cos`, `exp`, `log` and `fmod` floating-point only (§6.3.9). Given an integer they are refused with `E-TYPE-KIND`. Until 2026-09-16 they were not: `square 5` answered 5 instead of 25, and neither translation nor the run said anything — a quiet wrong answer. Write an integer power as multiplication, and convert with `cast f64 n` to compute in floating point.

examples/ch08/pow_fixed.low

```
module pow_fixed .
rem run: square_f 1.5
rem run: square_i 5

fn square_f input a f64 . output f64 .
do
  rem `pow` works correctly on floats
  return pow a 2.0 .
end

fn square_i input a u64 . output u64 .
  requires le a 1000 .
do
  rem an integer power is written as multiplication; the name says what it does
  return mul a a .
end
```

Output

```
$ lowentc --run square_f pow_fixed.low 1.5
square_f(1.5) = 2.25
$ lowentc --run square_i pow_fixed.low 5
square_i(5) = 25
```

A common misconception. Nested calls must always be parenthesised

examples/ch08/noparen.low

```
module noparen .
rem run: with_parens 3 4
rem run: without_parens 3 4

fn with_parens input a u64 . input b u64 . output u64 .
do
  return add 1 (mul a b) .
end
```

```
fn without_parens input a u64 . input b u64 . output u64 .
do
  rem without parentheses, the grouping comes from `mul` taking two arguments --- the sa
me 13
  return add 1 mul a b .
end
```

Output

```
$ lowentc --run with_parens noparen.low 3 4
with_parens(3, 4) = 13
$ lowentc --run without_parens noparen.low 3 4
without_parens(3, 4) = 13
```

Even without parentheses, the compiler groups a sentence by **the number of arguments each op takes**. `mul` takes two, so `add 1 mul a b` is the same as `add 1 (mul a b)`. This book still writes the parentheses. They show the grouping to readers who have not memorised argument counts, and they prevent the accident where a local name spelt like a builtin op changes the grouping (`count` in chapter 6).

8.7 This chapter's syntax at a glance

Shape	Meaning	Why
`add a (mul b c)`	prefix notation — no precedence	parentheses say the whole order of evaluation
`expr a + b * c`	infix island — `* /` bind tighter than `+ -`	only what school arithmetic taught goes infix
`expr (a + b) * c`	grouping inside the island	parentheses group; they are not values
`expr (lo le x) and (x le hi)`	comparisons mixed with logic	parentheses per comparison — no table to recall
`expr (twice a) + 1`	calling an op inside the island	without parentheses the call has no clear end (`E-EXPR-APP`)
`expr (mod a 10) + k`	an operation not in the table (`mod`, bitwise, `min`)	the island never grows — call it prefix
`expr 0 - a · (neg a)`	flip the sign	the island has no unary operators (`E-EXPR-UNARY`)
`and · or`	short-circuit — the right side is skipped once the left decides	put the condition that makes the right side safe first
`size_of u32`	bytes in one value of a type (at compile time)	generic code counts its cost honestly
`comptime (add 2 3)`	evaluate at compile time	folded branches are still type-checked

Table 8.2 — Expression syntax — shape · meaning · why it looks this way

Recap

The default for expressions is prefix notation, which has no precedence. The `expr` island allows only arithmetic, comparisons, `and` and `or` in infix; op calls inside it are parenthesised, and there are no unary operators. When mixing comparisons with `and` or `or`, parenthesise each comparison. Short-circuiting `and` and `or` can guard the right-hand side, and `size_of` and `comptime` are computed at translation time.

Part III — Data

9 Sequences — arrays and slices

In this chapter

9.1 Start and length together 110
9.2 Writing needs `mut slice` 111
9.3 Narrowing the window 112
9.4 Contracts remove bounds checks 114
9.5 Common mistakes 116
9.6 This chapter's syntax at a glance 119

What to know first

chapter 7, Flow · `for` walks a slice, and `guard` turns a condition into a fact
chapter 5, Ops · writes visible to the caller need a `proc`

Looking back

In chapter 7's `head_or_zero`, why was `index data 0` safe?

> **A.** Because the `guard ge (len data) 1 . else return 0 .` right before it handed the code below the fact that the slice was not empty. Code after a `guard` lives only in a world where the condition is true. This chapter covers what that slice is, and when bounds checks remain and when they disappear.

The need for this chapter, and its context

The most expensive defects in C come from people remembering separately how many items are at the address a pointer points to, and getting it wrong. That is the buffer overflow. Lowent has no pointers; pointing at several values is the job of a slice, which **carries its length with it**. Slices open Part III because nearly all data — strings, buffers, file contents — travels as slices.

By the end of this chapter

You will learn the difference between `array n t` and `slice t`, how to read with `len`, `index` and `for`, and that out-of-range access stops. You will pick up the rule that writing elements requires a `mut slice`, and how to narrow a window with `subslice`. You will also see the principle by which contracts remove bounds checks in the body, and that a string literal is a table that can be indexed directly.

The questions this chapter answers

1 Is there no separate string type?
2 Can `[3,4,5]` given to `--run` be used for a `slice u64`?

9.1 Start and length together

`slice t` is a contiguous run of values of type `t`. A slice value carries **its start address and its length** together, so nobody has to remember separately “how many are here”.

examples/ch09/basics.low

```
module basics .
rem run: head [9,8,7]
rem run: last4 [1,2,3,4]
rem run: total [10,20,30]
rem trap: at [1,2,3] 5

fn head input data slice u8 .
  output u8 .
  requires ge (len data) 1 .
do
  return index data 0 .
end

fn last4 input xs array 4 u8 . output u8 .
do
  return index xs 3 .
end

fn total input xs slice u8 . output u64 .
do
  var acc u64 be 0 .
  for x xs do
    set acc (add acc (widen u64 x)) .
  end
  return acc .
end

fn at input xs slice u8 . input i u64 . output u8 .
do
  return index xs i .
end
```

Output

```
$ lowentc --run head basics.low [9,8,7]
head([9,8,7]) = 9
  arg0 (written) = [9,8,7]
$ lowentc --run last4 basics.low [1,2,3,4]
last4([1,2,3,4]) = 4
  arg0 (written) = [1,2,3,4]
$ lowentc --run total basics.low [10,20,30]
total([10,20,30]) = 60
  arg0 (written) = [10,20,30]
$ lowentc --run at basics.low [1,2,3] 5
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

- `len data` is the number of elements, and `index data 0` is the first element. Numbering starts at 0.

- `array 4 u8` in `last4` is a sequence of exactly 4. An `array` in an input position means it receives a slice of length 4, and that length is checked on entry, like `requires eq (len xs) 4 ..`
- `for x xs do … end` walks the elements in order.
- Passing a slice of length 3 and 5 to `at` stops with `E-VM-BOUNDS`. It does not read someone else's memory.

An `array` writes its length **first**. The reverse order is rejected.

examples/ch09/array_order.low

```
module array_order .
rem expect: E-TYPE-ARRAY

fn last input xs array u8 4 . output u8 .
do
  return index xs 3 .
end
```

Output

```
$ lowentc --check array_order.low
array_order.low:4:0 E-TYPE-ARRAY: `array` is written `array <count> <type>` — the length
 first, as a literal (`array 4 u64`). The other order used to be read as a plain slice a
nd the length was dropped silently
```

According to the diagnostic, the old tool read `array u8 4` as a plain slice and silently dropped the length — a place where a written promise vanished. Today `array` is used only in op input positions; in outputs, locals and struct fields, write `slice` with a contract.

> **Q. Is there no separate string type?**
>
> **A.** There is not. A string literal `"hello"` is a `slice u8`. Questions such as how many characters it has or whether it is valid UTF-8 are answered by library ops, not by a type (chapter 33). It is a choice to avoid the defects that start the moment bytes and characters are treated as the same thing.

9.2 Writing needs `mut slice`

To write the elements of a slice, the slice must be `mut`. And changing bytes the caller passed is visible to the caller, so it must be a `proc` (chapter 5).

examples/ch09/fill.low

```
module fill .
rem run: fill_two [0,0,5]

proc fill_two input xs mut slice u8 . output u64 . effects state .
  requires ge (len xs) 2 .
do
  set (index xs 0) 10 .
  set (index xs 1) 20 .
  var s u64 be 0 .
  for x xs do
```

```
    set s (add s (widen u64 x)) .
  end
  return s .
end
```

Output

```
$ lowentc --run fill_two fill.low [0,0,5]
fill_two([10,20,5]) = 35
  arg0 (written) = [10,20,5]
```

The `fill_two([10,20,5])` the VM shows is the argument's state after the op ended; the caller's bytes changed. Writing without `mut` is rejected.

examples/ch09/readonly.low

```
module readonly .
rem expect: E-TYPE-MUT

fn poke input xs slice u8 . output u64 .
do
  set (index xs 0) 1 .
  return 0 .
end
```

Output

```
$ lowentc --check readonly.low
readonly.low:6:0 E-TYPE-MUT: writing an element of a slice that is not declared `mut` (a
 shared slice is read-only)
```

`mut` is a permission on the slice's **elements**. It is a different question from whether the name holding the slice was made with `let` (chapter 6). It is followed by the rule that a slice whose elements can be written may be borrowed in only one place at a time (chapter 12).

9.3 Narrowing the window

`subslice s from to` makes a new slice pointing from position `from` of `s` up to, but not including, `to`. It copies no bytes; it just places another, narrower window over the same bytes.

examples/ch09/windows.low

```
module windows .
rem run: middle_sum [1,2,3,4,5]
rem run: prefix_char 4

fn middle_sum input xs slice u8 . output u64 .
  requires ge (len xs) 3 .
do
  let mid slice u8 be subslice xs 1 (sub (len xs) 1) .
  var s u64 be 0 .
  for x mid do
    set s (add s (widen u64 x)) .
  end
```

```
  return s .
end

fn prefix_char input k u64 . output u8 .
  requires lt k 11 .
do
  return index "/api/users/" k .
end
```

Output

```
$ lowentc --run middle_sum windows.low [1,2,3,4,5]
middle_sum([1,2,3,4,5]) = 9
  arg0 (written) = [1,2,3,4,5]
$ lowentc --run prefix_char windows.low 4
prefix_char(4) = 47
```

`middle_sum` sums the middle elements, leaving out both ends. `prefix_char` indexes a string literal directly, because a literal is a `slice u8`.

In practice. From an `if` chain to one table

An op returning a constant per index is easy to write as a chain like `if eq k 0 . do return 47 . end`. The emitted C then has one comparison and branch per character. Indexing a string literal becomes one static table and one index. In a benchmark that reads HTTP requests, replacing such a chain with a table lookup brought branch mispredictions down to the level of hand-written C and cut run time by about 25%. The answer did not change by a single bit.

examples/ch09/table.low

```
module table .
rem run: prefix_char 4
rem run: prefix_chain 4

rem a string literal is a slice u8 and can be indexed as it is
fn prefix_char input k u64 . output u8 .
  requires lt k 11 .
do
  return index "/api/users/" k .
end

rem the same job as an if chain costs one comparison and one branch per character
fn prefix_chain input k u64 . output u8 . do
  if eq k 0 . do return 47 . end
  if eq k 1 . do return 97 . end
  if eq k 2 . do return 112 . end
  if eq k 3 . do return 105 . end
  if eq k 4 . do return 47 . end
  return 0 .
end
```

Output

```
$ lowentc --run prefix_char table.low 4
prefix_char(4) = 47
```

```
$ lowentc --run prefix_chain table.low 4
prefix_chain(4) = 47
```

The two ops give the same answer. Looking with `--emit-c`, `prefix_char` is one string table and one index, while `prefix_chain` is a chain of comparisons and `goto`s. In this edition, though, the index check in `prefix_char` remains — the analysis does not connect the literal's length 11 with the contract `lt k 11`.

9.4 Contracts remove bounds checks

An `index` bounds check remains only where it is not proven. Write the length condition as `requires`, and the check happens **once** on entry to the op while the index checks in the body disappear.

examples/ch09/bounds.low

```
module bounds .
rem run: sum_first [3,4,5,6] 3

fn sum_first input a slice u8 . input n u64 . output u64 .
  requires le n (len a) .
do
  var s u64 be 0 .
  var i u64 be 0 .
  while lt i n . do
    set s (add s (widen u64 (index a i))) .
    set i (add i 1) .
  end
  return s .
end
```

Output

```
$ lowentc --run sum_first bounds.low [3,4,5,6] 3
sum_first([3,4,5,6], 3) = 12
  arg0 (written) = [3,4,5,6]
```

With `requires le n (len a) .`, `i` inside `while lt i n .` is always less than `len a`. The compiler's interval analysis works that out and removes the check in `index a i`. The rules behind this reasoning are proven in Coq, and an independent checker re-verifies the arithmetic evidence the compiler leaves at every removed check (chapter 41).

One thing to watch: a count `n` uses `le`, but an **index** itself uses `lt`. `requires le i (len a) .` allows `i = len a`, which is one past the end.

> **A common misconception. Putting `len` in a loop condition recounts every time**
>
> `len s` is neither a call nor a memory read. It just takes the length field out of the slice value `{start, length}`. Writing elements does not change the length; the only way the length changes is putting a different slice into the name. In fact, storing the length ahead as `let m u64 be len xs .` and then replacing `xs` with a shorter slice leaves `m` as a **stale length** that can go out of range. Storing it ahead is a choice of meaning, not an optimisation.

examples/ch09/stale_len.low

```
module stale_len .
rem run: fresh [1,2,3,4]
rem trap: stale [1,2,3,4]

rem len stays in the condition --- each round reads the current length
fn fresh input xs slice u8 . output u64 . do
  var s slice u8 be xs .
  var t u64 be 0 .
  var i u64 be 0 .
  while lt i (len s) . do
    set t (wrap_add t (widen u64 (index s i))) .
    set s (subslice s 0 2) .
    set i (add i 1) .
  end
  return t .
end

rem the length was stored ahead --- m stays even when s gets shorter
fn stale input xs slice u8 . output u64 . do
  var s slice u8 be xs .
  let m u64 be len s .
  var t u64 be 0 .
  var i u64 be 0 .
  while lt i m . do
    set t (wrap_add t (widen u64 (index s i))) .
    set s (subslice s 0 2) .
    set i (add i 1) .
  end
  return t .
end
```

Output

```
$ lowentc --run fresh stale_len.low [1,2,3,4]
fresh([1,2,3,4]) = 3
  arg0 (written) = [1,2,3,4]
$ lowentc --run stale stale_len.low [1,2,3,4]
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

`fresh` reads the current length each time round, so when `s` shrinks to two elements the loop stops there too. `stale` stored the first length 4 in `m`, so on the third round it tries to read element 2 of the two-element `s` and stops. `len` reads the current value every time, and that is what keeps it safe.

> **Q. Can `[3,4,5]` given to `--run` be used for a `slice u64`?**
>
> **A.** Bracketed `--run` arguments are sequences of bytes. That is why this book's runnable examples take slices as `slice u8`. Passing one to a `slice u64` stops the VM, because the byte count is not a multiple of 8. Slices of wider elements are usually built inside the program and passed along.

9.5 Common mistakes

Almost every slice mistake comes down to **being off by one**. Where C would read someone else's memory and carry on, Lowent stops.

Counter-example. Looping one step too far with `le`

examples/ch09/mistake_offbyone.low

```
module mistake_offbyone .
rem trap: total [1,2,3]

fn total input xs slice u8 . output u64 .
do
  var t u64 be 0 .
  var i u64 be 0 .
  rem ✘ with `le` it also runs when i is 3 --- a slice of length 3 has indexes 0, 1, 2 o
nly
  while le i (len xs) . do
    set t (add t (widen u64 (index xs i))) .
    set i (add i 1) .
  end
  return t .
end
```

Output

```
$ lowentc --run total mistake_offbyone.low [1,2,3]
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

A slice of length 3 has indexes 0, 1 and 2. `le i (len xs)` is still true when `i` is 3, so reading the fourth cell stops with `E-VM-BOUNDS`. Because indexes start at 0, the right condition is "less than the count" — `while lt i (len xs) .`. If you are walking every element, `for x xs do` uses no index at all and removes the mistake completely.

Counter-example. Reading with brackets, as in `xs[0]`

examples/ch09/mistake_cindex.low

```
module mistake_cindex .
rem expect: E-CHAR

fn first input xs slice u8 . output u8 .
do
  rem ✘ there is no bracket indexing --- read with `index xs 0`
  return xs[0] .
end
```

Output

```
$ lowentc --check mistake_cindex.low
7:12 E-CHAR: unexpected character
7:14 E-CHAR: unexpected character
7:12 E-FORM-UNEXPECTED: unexpected token in form
```

There is no bracket indexing. Reading an element is `index xs 0`, writing it is `set (index xs 0) v` .. Using names instead of symbols makes reading, writing and the out-of-range stop all the same shape of form, with fewer symbols to remember.

Counter-example. Reading the first element of what may be an empty slice

examples/ch09/mistake_empty.low

```
module mistake_empty .
rem trap: first []

fn first input xs slice u8 . output u8 .
do
  rem ✘ an empty slice has no element 0
  return index xs 0 .
end
```

Output

```
$ lowentc --run first mistake_empty.low []
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

It is easy to forget that an input may be empty. Index 0 exists only when there is at least one element. Filter first, then read.

examples/ch09/empty_fixed.low

```
module empty_fixed .
rem run: first_or [] 0
rem run: first_or [7,8] 0

fn first_or input xs slice u8 . input fallback u8 . output u8 .
do
  rem leave here if empty --- below this line, having at least one element is a fact
  guard gt (len xs) 0 . else return fallback .
  return index xs 0 .
end
```

Output

```
$ lowentc --run first_or empty_fixed.low [] 0
first_or([], 0) = 0
  arg0 (written) = []
$ lowentc --run first_or empty_fixed.low [7,8] 0
first_or([7,8], 0) = 7
  arg0 (written) = [7,8]
```

Below the `guard`, “not empty” is a fact, so `index xs 0` is safe — and the compiler uses the same fact to remove the bounds check (chapter 7).

Counter-example. Swapping the start and end of `subslice`

examples/ch09/mistake_subrev.low

```
module mistake_subrev .
rem trap: tail_len [1,2,3,4]

fn tail_len input xs slice u8 . output u64 .
do
  rem ✘ the start (3) is after the end (1) --- `subslice s from to` needs from ≤ to ≤ le
n
  return len (subslice xs 3 1) .
end
```

Output

```
$ lowentc --run tail_len mistake_subrev.low [1,2,3,4]
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: subslice out of bounds (panic)
```

`subslice s from to` needs `from ≤ to ≤ len s`. Swapping them does not give an empty window; it stops — because a miscalculated boundary that quietly became an empty result would only surface much later.

A common misconception. `subslice s 1 3` is the three cells from 1 to 3

examples/ch09/subslice_end.low

```
module subslice_end .
rem run: middle [10,20,30,40,50]

fn middle input xs slice u8 . output u64 .
do
  rem from index 1 up to (not including) 3 --- indexes 1 and 2, two elements
  return len (subslice xs 1 3) .
end
```

Output

```
$ lowentc --run middle subslice_end.low [10,20,30,40,50]
middle([10,20,30,40,50]) = 2
  arg0 (written) = [10,20,30,40,50]
```

The end index is **not included**: indexes `1` and `2`, two cells. With this rule the length is simply `to - from`, and `subslice s 0 k` and `subslice s k (len s)` split the slice with no cell overlapping or missing. That is why most languages use the same rule (a half-open range).

Counter-example. Taking a string literal as `mut slice` and changing it

examples/ch09/mistake_litwrite.low

```
module mistake_litwrite .
rem expect: E-TYPE-ARGMUT

rem ✘ taking a string literal as mut slice and changing it; a literal is read-only bytes
 baked into the program
fn poke output u8 . do
```

```
  let buf mut slice u8 . be "abc" .
  set (index buf 0) 65 .
  return index buf 0 .
end
```

Output

```
$ lowentc --check mistake_litwrite.low
mistake_litwrite.low:6:0 E-TYPE-ARGMUT: a string LITERAL was bound to a name declared `m
ut`. A literal is bytes baked into the program, not a place that can be written: the VM
used to change them while the native build did not, so the same program gave two answers
, and a library op writing there killed the native build. Take bytes you will change fro
m `alloc_bytes` or from the caller's buffer, and copy the literal into them
```

A string literal is bytes baked into the program, not a place to change. So binding it to a name declared `mut`, or passing it where a `mut` place is taken, is refused with `E-TYPE-ARGMUT`. Until 2026-09-16 both passed, and when run the VM returned 65 while native code returned 97 — the same program with two answers — and a standard-library op writing those bytes killed the native build with a segmentation fault. Take bytes you will change from `alloc_bytes` or from the caller's buffer, and copy the literal into them.

9.6 This chapter's syntax at a glance

Shape	Meaning	Why
`slice u8`	a run of `u8` values (start + length)	no separate count to carry around
`mut slice u8`	a run whose elements may be written	writability is visible in the type
`input xs array 4 u8 .`	take a run of exactly 4 (input position only)	length first — checked at entry
`len xs`	number of elements	just reads the length field; no cost
`index xs i`	element `i` (from 0)	stops when out of range — never reads someone else's memory
`set (index xs i) v .`	write element `i`	needs a `mut slice` and a `proc`
`for x xs do … end`	each element in turn	no room for index mistakes
`subslice xs from to`	a window from `from` up to, not including, `to` (no copy)	half-open — the length is `to - from`
`"hello"`	a literal of type `slice u8` — indexable as is	there is no separate string type
`requires le n (len xs) .`	a length condition as a contract	bounds checks in the body are removed

Table 9.1 — Slice syntax — shape · meaning · why it looks this way

Recap

A slice carries its start and length together. `len` takes the length field, `index` stops when out of range, and `for` walks the elements. `array n t` writes the length first and is used in input positions. Writing elements needs a `mut slice` and a `proc`. `subslice` narrows the window without copying. Writing the length condition as a contract removes bounds checks in the body, and a string literal is a table that can be indexed directly.

10 Aggregates — `struct` and `enum`

In this chapter

10.1 `struct` — a collection of named fields 121
10.2 `enum` — one of several 122
10.3 Variants are closed with full stops 124
10.4 Nothing may contain itself 124
10.5 Common mistakes 126
10.6 This chapter's syntax at a glance 129

What to know first

chapter 7, Flow · `match` must cover every case
chapter 9, Sequences · a slice carries its length

Looking back

Besides the exhaustiveness check, what did chapter 7 give as the advantage of `match` over an `if` chain?

A. What happens when cases grow. Add a variant to an enum and every `match` over it stops at translation and points to what must be fixed, while an `if` chain lets the new case slide silently into its last `else`. This chapter makes such enums.

The need for this chapter, and its context

Numbers and slices alone cannot write shapes like "a point", "a segment", or "a circle or a rectangle or a dot". You need a collection of named fields (`struct`) and one-of-several (`enum`) for data to take the shape of the problem. In particular, the pairing of an `enum` carrying values with `match` is Lowent's main replacement for inheritance, and `option` and `result` (chapter 11) and traits (chapter 23) all build on it.

> ### By the end of this chapter
>
> You will learn to declare a `struct`, make one with `make` and read it with `field`, and why there is no glued dot. You will see that the variants of an `enum` can carry values, how to split them with `match` while binding those values to names, and why variants must be closed with full stops. You will also learn why a type that contains itself is rejected, and how to link tree structures by number.

> ### The questions this chapter answers
>
> 1. Can a field be named `to` or `in`?
> 2. Isn’t linking by number less convenient than linking by pointer?

10.1 `struct` — a collection of named fields

A `struct` is a collection of named fields, each with its own type. The declaration’s body is `do` … `end`, and each field is closed with a full stop.

examples/ch10/points.low

```
module points .
rem run: demo_dist 3 4
rem run: demo_move 5

struct point do
  x i64 .
  y i64 .
end

struct segment do
  start point .
  stop point .
end

fn manhattan input s segment . output i64 .
do
  let dx i64 be sub (field s stop x) (field s start x) .
  let dy i64 be sub (field s stop y) (field s start y) .
  return add (abs dx) (abs dy) .
end

fn demo_dist input dx i64 . input dy i64 . output i64 .
do
  let a point be make point do x 1 . y 1 . end .
  let b point be make point do x (add 1 dx) . y (add 1 dy) . end .
  return manhattan (make segment do start a . stop b . end) .
end

fn moved input p point . input k i64 . output point .
do
  return make point do x (add (field p x) k) . y (field p y) . end .
end
```

```
fn demo_move input k i64 . output i64 .
do
  let p point be make point do x 10 . y 0 . end .
  let q point be moved p k .
  return field q x .
end
```

Output

```
$ lowentc --run demo_dist points.low 3 4
demo_dist(3, 4) = 7
$ lowentc --run demo_move points.low 5
demo_move(5) = 15
```

- `make point do x 1 . y 1 . end` makes a value. **Every field must be filled** when making it. No unfilled field quietly becomes 0.
- `field s stop x` reads the `stop` field of `s` and the `x` field inside it. With several steps, it goes down one field at a time from the left.
- A field's type may be another `struct`. `segment` holds two `point`s.
- `moved` builds and returns a **new value** with one field changed. The original stays as it was.

There is no glued dot like `p.x`. Dots are already used for module names (`allocs.byte_allocator`) and variant names (`shape.dot`). If looking inside a value were written with a dot too, what `a.b` means would only be settled after looking up whether `a` is a value or a module. The `field` form settles it the moment you read it.

> **Q. Can a field be named `to` or `in`?**
>
> **A.** No. `to` and `in` were infix spellings for reading fields in the old grammar (`a to b`) and are now removed words. A spelling that is a word cannot be a name, so it is rejected with `E-VOCAB-REMOVED`. That is why this example uses `start` and `stop` rather than `from` and `to`.

`field` is both a place to read and a place to write. A field of a value received as `mut` is changed with `set (field p x) 3 .` — reading and writing use the same spelling. The rules for borrowing a value to change it are in chapter 12.

10.2 `enum` — one of several

An `enum` is one of several variants. A variant can carry values, written as `<field name> <type>` pairs.

examples/ch10/shapes.low

```
module shapes .
rem run: demo_area 0
rem run: demo_area 1
rem run: demo_area 2

enum shape do
  circle r u32 .
  rect w u32 h u32 .
  dot .
```

```
end

fn area input s shape . output u32 .
do
  match s do
    case circle r . do return mul 3 (mul r r) . end
    case rect w h . do return mul w h . end
    case dot . do return 0 . end
  end
end

fn demo_area input k u8 . output u32 .
do
  if eq k 0 . do return area (shape.circle 2) . end
  if eq k 1 . do return area (shape.rect 3 5) . end
  return area shape.dot .
end
```

Output

```
$ lowentc --run demo_area shapes.low 0
demo_area(0) = 12
$ lowentc --run demo_area shapes.low 1
demo_area(1) = 15
$ lowentc --run demo_area shapes.low 2
demo_area(2) = 0
```

- `circle r u32 .` carries one value named `r`, and `rect w u32 h u32 .` carries two. `dot .` carries nothing.
- A value is made with the **variant name**, as in `shape.circle 2`.
- `case rect w h .` in a `match` binds the two carried values to `w` and `h` when the variant is `rect`.

However many variants there are, the `match` must cover them all. Leaving out `dot` is rejected.

examples/ch10/missing_case.low

```
module missing_case .
rem expect: E-MATCH-INEXHAUSTIVE

enum shape do
  circle r u32 .
  rect w u32 h u32 .
  dot .
end

fn corners input s shape . output u32 .
do
  match s do
    case circle r . do return 0 . end
    case rect w h . do return 4 . end
  end
end
```

Output

```
$ lowentc --check missing_case.low
missing_case.low:12:0 E-MATCH-INEXHAUSTIVE: this `match` does not handle every variant o
f the enum — add the missing `case` (that exhaustiveness is what a `match` buys you over
 an if-chain: add a variant later and the compiler finds every place that must change),
or add a `case _ .` wildcard to cover the rest
```

This check is the greatest value of an `enum`. When a `triangle` variant is added later, every `match` over `shape`, `area` and `corners` included, stops at translation. Nobody has to remember the places to fix.

To ask only which variant without a `match`, `isa n lit` gives a `bool`.

10.3 Variants are closed with full stops

A newline is not a closer (chapter 3). So writing variants one per line without full stops runs several variants together into one.

examples/ch10/enum_dot.low

```
module enum_dot .
rem expect: E-ENUM-DOT

enum color do
  red
  green
  blue .
end
```

Output

```
$ lowentc --check enum_dot.low
enum_dot.low:6:0 E-ENUM-DOT: this enum variant is not closed with `.`, so the NEXT line
was read as part of it — a newline does not close a form (RFC-0103). Close every variant
: `red .` · `green .`. Without the dot `red green blue` is ONE variant `red` whose paylo
ad field `green` has a type `blue`
```

`red green blue` nearly became one variant's name plus field pairs. The compiler rejects it and tells you to close each variant, as in `red .`.

> **A common misconception. An `enum` is a named integer, like C's enumerations**
>
> C's `enum` gives names to integer constants and mixes freely with integers. Lowent's `enum` is a **type**. It does not mix with integers, each variant can carry values of a different shape, and splitting it must cover every variant. It is closer to Rust's `enum` or the algebraic data types of the ML family.

10.4 Nothing may contain itself

A variant that holds its own type **by value** would have infinite size.

examples/ch10/infinite.low

```
module infinite .
rem expect: E-ENUM-INFINITE

enum expr_tree do
  lit v u32 .
  add l expr_tree r expr_tree .
end
```

Output

```
$ lowentc --check infinite.low
6:0 E-ENUM-INFINITE: an enum variant cannot embed its OWN type BY VALUE — the size would
 be infinite. Recursion must go through indirection: an index into a node arena (e.g. `u
32`) or an `owned` box (RFC-0080 §4.2)
6:0 E-ENUM-INFINITE: an enum variant cannot embed its OWN type BY VALUE — the size would
 be infinite. Recursion must go through indirection: an index into a node arena (e.g. `u
32`) or an `owned` box (RFC-0080 §4.2)
```

The same goes for `struct`. Holding itself as a field, directly or through other types, is rejected with `E-STRUCT-CYCLE`. When you need a tree-like structure, link it **by number** as the diagnostic suggests. Put the nodes in a slice and point at children by their position in that slice.

examples/ch10/tree.low

```
module tree .

enum node do
  lit v u32 .
  add l u32 r u32 .
  mul l u32 r u32 .
end

fn eval input nodes slice node . input i u32 . output u32 .
do
  let n node be index nodes i .
  match n do
    case lit v . do return v . end
    case add l r . do return add (eval nodes l) (eval nodes r) . end
    case mul l r . do return mul (eval nodes l) (eval nodes r) . end
  end
end

fn is_leaf input n node . output bool .
do
  return isa n lit .
end
```

Output

```
$ lowentc --check tree.low
== check: ok ==
```

The `l` and `r` of `add l u32 r u32 .` are not nodes but positions in `nodes`. `eval` finds children by position and recurses. The size is fixed, and the `index` that follows a position still gets its bounds check. The storage for such node slices — regions — is covered in chapter 18.

> **Q. Isn't linking by number less convenient than linking by pointer?**
>
> **A.** In one way it is. If a node is removed and its number reused, whoever held the old number sees the wrong node. The standard library's `pool` catches this at run time with **generational handles** that carry a generation count along with the number (chapter 35). What you gain is large: numbers are safe to copy, can be written straight to a file, and keep nodes together in one slice, which suits the cache.

10.5 Common mistakes

> **Counter-example. Leaving a field out of `make`**

examples/ch10/mistake_missfield.low

```
module mistake_missfield .
rem expect: E-TYPE-FIELD

struct point do
  x u64 .
  y u64 .
end

fn origin_x output u64 .
do
  rem ✘ `y` is not filled in --- `make` must fill every field
  let p point be make point do x 1 . end .
  return field p x .
end
```

Output

```
$ lowentc --check mistake_missfield.low
mistake_missfield.low:12:0 E-TYPE-FIELD: a declared field is missing from this `make` (e
very field must be given)
```

C fills missing struct fields with 0, and some languages insert a default. Lowent rejects it with `E-TYPE-FIELD`, because the source cannot tell whether the missing field was intended or forgotten. If you want 0, write `y 0 .` — a written 0 is obviously intended. A misspelt field name produces the same code, saying "no such field".

> **Counter-example. Passing a value-less variant as `Type.variant`**

examples/ch10/mistake_unitvariant.low

```
module mistake_unitvariant .
rem run: pick_bare
rem run: pick_qualified

enum light do
  red .
  green .
end
```

```
fn code input c light . output u64 .
do
  match c do
    case red . return 1 .
    case green . return 2 .
  end
end

fn pick_bare output u64 .
do
  rem a variant that carries no value is written by its bare name
  return code green .
end

fn pick_qualified output u64 .
do
  rem ✘ in this edition, passing `light.green` stops while running
  return code light.green .
end
```

Output

```
$ lowentc --run pick_bare mistake_unitvariant.low
pick_bare() = 2
$ lowentc --run pick_qualified mistake_unitvariant.low
pick_qualified() = 2
```

A variant that carries values is built with the type name in front, as in `shape.circle 2`, and this book writes a variant carrying nothing by **its name alone** (`green`). Both are the same value — `pick_bare` and `pick_qualified` both answer 2. Until 2026-09-16 the tool lowered a value-less variant written as `light.green` to a **different representation** (a record with a tag), so it stopped at run time with `E-VM-TYPE`. Now both lower to the same index. Prefer the shorter spelling, and qualify where the enum would otherwise be hard to tell.

Counter-example. Comparing two structs with `eq`

examples/ch10/mistake_eqstruct.low

```
module mistake_eqstruct .
rem expect: E-TYPE-KIND

struct point do
  x u64 .
  y u64 .
end

fn same output bool .
do
  let p point be make point do x 1 . y 2 . end .
  let q point be make point do x 1 . y 2 . end .
  rem ✘ compares two structs with `eq` --- `eq` compares numbers and booleans
  return eq p q .
end
```

Output

```
$ lowentc --check mistake_eqstruct.low
mistake_eqstruct.low:14:0 E-TYPE-KIND: `eq` / `ne` compare numbers and booleans, not STR
UCTS. What "equal" means for a struct differs by type — every field, or only the identif
ying one? — so the language does not guess: write an op that says it (field by field). T
his used to pass `--check` and stop at run time with `E-VM-TYPE`
```

`eq` compares numbers and booleans. What it means for two structs to be “equal” differs by type — must every field match, or only an identifying one? So it is rejected at translation time with `E-TYPE-KIND`. Write down what equality means as an op.

examples/ch10/eqstruct_fixed.low

```
module eqstruct_fixed .
rem run: same

struct point do
  x u64 .
  y u64 .
end

rem spell out field by field what "equal" means --- it differs from type to type
fn point_eq input a point . input b point . output bool .
do
  return expr ((field a x) eq (field b x)) and ((field a y) eq (field b y)) .
end

fn same output bool .
do
  let p point be make point do x 1 . y 2 . end .
  let q point be make point do x 1 . y 2 . end .
  return point_eq p q .
end
```

Output

```
$ lowentc --run same eqstruct_fixed.low
same() = 1
```

Counter-example. Believing that putting a value under another name makes a copy

examples/ch10/mistake_alias.low

```
module mistake_alias .
rem run: copy_then_change

struct point do
  x u64 .
  y u64 .
end

fn copy_then_change output u64 .
do
  let p point be make point do x 1 . y 2 . end .
```

```
  rem ✘ meant to make a copy --- in this edition q refers to the same place as p
  var q point be p .
  set (field q x) 99 .
  rem p is a let, and yet this gives 99
  return field p x .
end
```

Output

```
$ lowentc --run copy_then_change mistake_alias.low
copy_then_change() = 1
```

`var q point be p .` builds **a new value with `p`'s fields copied**. So changing `q`'s field to 99 leaves `p` at 1. A value without ownership is copied; a value with ownership is moved (chapter 19). Until 2026-09-16 the tool made an alias to the same place, and `p` changed too — the `let` promise broke there. Lowering now copies the fields.

examples/ch10/alias_fixed.low

```
module alias_fixed .
rem run: copy_then_change

struct point do
  x u64 .
  y u64 .
end

fn copy_then_change output u64 .
do
  let p point be make point do x 1 . y 2 . end .
  rem when you need a new value, build it with make --- copy the fields one by one
  var q point be make point do x (field p x) . y (field p y) . end .
  set (field q x) 99 .
  return field p x .
end
```

Output

```
$ lowentc --run copy_then_change alias_fixed.low
copy_then_change() = 1
```

This is also why `moved` in this chapter's first example returns a new value instead of changing a field. Making a new value rather than modifying one sidesteps aliasing altogether.

10.6 This chapter's syntax at a glance

Shape	Meaning	Why
`struct point do x u64 . y u64 . end`	a bundle of named fields	one line per field, name and type
`make point do x 1 . y 2 . end`	build a value — fill every field	no field silently becomes 0

`field p x · field s stop x`	read a field · walk down several levels	no glued dot — the meaning is fixed as you read
`set (field p x) 3 .`	write a field (of a value received `mut`)	reading and writing are spelt the same
`enum shape do dot . circle r u32 . end`	one of several — variants may carry values	each variant is closed with a stop
`shape.circle 2 · dot`	build a variant that carries a value · one that carries none	the variant name is the constructor
`match s do case circle r . … end`	split on variants and bind their values	every variant must be covered
`isa s circle`	is it that variant (`bool`)	for asking without taking values out
trees linked by index (`1 u32 · r u32`)	slice indexes instead of containing itself	the size is fixed and indexes are bounds-checked

Table 10.1 — Struct and enum syntax — shape · meaning · why it looks this way

Recap

A `struct` is made with `make`, filling every field, and read and written with `field`. There is no glued dot. The variants of an `enum` can carry values, are made as `<type>.<variant>`, and are split with a `match` covering every variant. Variants are closed with full stops. A type holding itself by value has infinite size and is rejected; tree structures are linked by number.

11 Types that hold answers — `option` and `result`

In this chapter

11.1 `option` — a value, or none 131
11.2 `result` and the `errors` clause 133
11.3 `try` — passing failure upwards 135
11.4 Crossing between the two channels 135
11.5 Three ways to report failure 136
11.6 Combining and nesting patterns 137
11.7 Common mistakes 139
11.8 This chapter's syntax at a glance 143

What to know first

chapter 7, Flow · `guard` and `panic`
chapter 10, Aggregates · enums carrying values, and `match`

Looking back

What does `case rect w h .` in chapter 10 do? And what happens when a `match` leaves out a variant?

> **A.** When the variant is `rect`, it binds the two values the variant carries to `w` and `h`. Leaving out a variant is rejected with `E-MATCH-INEXHAUSTIVE`. The `option` and `result` of this chapter behave like two-variant enums the language made in advance — a value is there or not, it succeeded or failed.

The need for this chapter, and its context

The habit of signalling failure with a special value such as −1 or a null pointer makes it impossible to tell from the source whether "this −1 is an error or just −1". And forgetting the check goes unnoticed. `option` and `result` move that question into the type. This chapter sits in the middle of Part III because nearly every op that takes data out — a lookup, a conversion, a parser — returns "may be missing" or "may fail".

By the end of this chapter

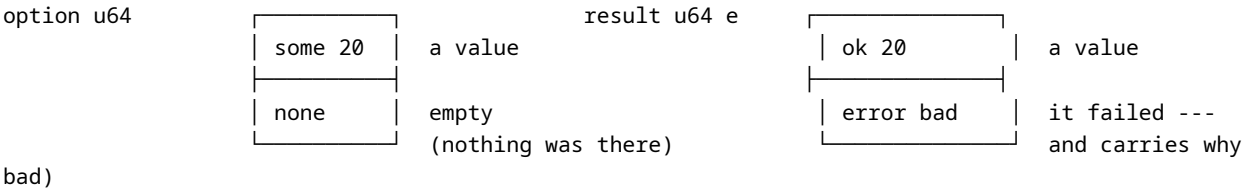
You will learn to make an `option` with `some` and `none`, and to use it by asking and taking out, by giving a fallback with `value_or`, or by splitting with `match`. You will see that taking out without checking stops. You will also see that `result` pairs with an `errors` clause, that `try` passes a failure upwards, and that the `else_none` and `else_error` tails cross between the two. Finally you will settle the criterion that separates the three ways of reporting failure — `result`, `option` and `panic` — and see or-patterns, nested patterns and `match` folded at translation time.

The questions this chapter answers

1 If I put an expensive computation in `value_or`'s default, is it computed every time?

11.1 `option` — a value, or none

`option t` is either a value of `t` (`some v`) or nothing (`none`). Together with `result`, seen later, it helps to draw them as **two kinds of box** that hold a value.



`none` is not something strange but the normal answer "there was nothing", and `error` is the answer "what I tried failed". Either way, the value inside cannot be used **before the box is opened**.

examples/ch11/lookup.low

```
module lookup .
rem run: find 2
rem run: find 7
rem run: find_or 7
rem run: find_asked 2
rem run: find_match 7
rem trap: find_raw 7

fn find input k u8 . output option u8 .
do
  guard lt k 3 . else return none .
  return some (mul k 10) .
end

fn find_or input k u8 . output u8 .
do
  return value_or (find k) 99 .
end

fn find_asked input k u8 . output u8 .
do
  let r option u8 be find k .
  guard is_some r . else return 255 .
  return some_value r .
end

fn find_match input k u8 . output u8 .
do
  match find k do
    case some v . do return v . end
    case none . do return 0 . end
  end
end

fn find_raw input k u8 . output u8 .
do
  return some_value (find k) .
end
```

Output

```
$ lowentc --run find lookup.low 2
find(2) = some 20
$ lowentc --run find lookup.low 7
find(7) = none
$ lowentc --run find_or lookup.low 7
find_or(7) = 99
$ lowentc --run find_asked lookup.low 2
find_asked(2) = 20
$ lowentc --run find_match lookup.low 7
find_match(7) = 0
$ lowentc --run find_raw lookup.low 7
== ir diagnostics (1) ==
0:0 E-VM-NONE: some_value of none (panic)
```

`find`, the producer, wraps values with `return none .` and `return some (mul k 10) .`, and the VM shows the results as `some 20` and `none`. The receiver can use it three ways.

- `find_or` — `value_or (find k) 99` gives the value if there is one, and 99 otherwise.
- `find_asked` — asks first with `guard is_some r . else …` and takes the value out with `some_value r`.
- `find_match` — splits with `match` into `case some v .` and `case none .`. The two arms cover every case.

`find_raw` takes the value out without asking. Translation passes, but execution stops at 7, which has no value (`E-VM-NONE`). Taking a value out is a **partial operation**. On which paths a value exists is something the author knows and the processor cannot always know, so translation does not block it. Instead it is not silent when wrong — it does not hand out 0 and carry on.

> **Q. If I put an expensive computation in `value_or`'s default, is it computed every time?**
>
> **A.** No. The default is computed **only when there is no value**. `value_or (some 7) (div 1 0)` is 7, and no division by zero happens. So you may put a computation that can fail in the default. This behaviour was once the other way round and was fixed to match the specification.

11.2 `result` and the `errors` clause

`result t e` is either a successful value (`ok v`) or an error (`error <variant>`). The error type `e` is usually an `enum`. And an op that returns a `result` writes **when it produces which error** in its `errors` clause.

examples/ch11/halve.low

```
module halving .
rem run: halve 100
rem run: halve 250
rem run: halve 101
rem run: halve_plus_one 100
rem run: halve_plus_one 250
rem run: halve_or_zero 250

enum io_error do
  too_big .
  odd .
end

fn halve input a u8 . output result u8 io_error .
  errors too_big gt a 200 .
  errors odd ne (mod a 2) 0 .
do
  guard le a 200 . else return error too_big .
  guard eq (mod a 2) 0 . else return error odd .
  return ok (div a 2) .
end

fn halve_plus_one input a u8 . output result u8 io_error .
  errors too_big gt a 200 .
  errors odd ne (mod a 2) 0 .
do
  let v u8 be try halve a .
  return ok (add v 1) .
end

fn halve_or_zero input a u8 . output u8 .
```

```
do
  let r result u8 io_error be halve a .
  guard not (is_error r) . else return 0 .
  return ok_value r .
end
```

Output

```
$ lowentc --run halve halve.low 100
halve(100) = ok 50
$ lowentc --run halve halve.low 250
halve(250) = err too_big
$ lowentc --run halve halve.low 101
halve(101) = err odd
$ lowentc --run halve_plus_one halve.low 100
halve_plus_one(100) = ok 51
$ lowentc --run halve_plus_one halve.low 250
halve_plus_one(250) = err too_big
$ lowentc --run halve_or_zero halve.low 250
halve_or_zero(250) = 0
```

`halve`’s head lists two errors. `errors too_big gt a 200 .` is the promise “if `a` is greater than 200, produce `too_big`”. This clause is a **contract on the way out**. Returning normally while the condition holds, or returning an error not written, is a contract violation. Code that returns an error not written is rejected at translation.

examples/ch11/undeclared.low

```
module undeclared .
rem expect: E-ERR-UNDECLARED

enum parse_error do
  empty .
  too_long .
end

fn first_byte input s slice u8 . output result u8 parse_error .
  errors empty eq (len s) 0 .
do
  guard gt (len s) 0 . else return error empty .
  guard le (len s) 16 . else return error too_long .
  return ok (index s 0) .
end
```

Output

```
$ lowentc --check undeclared.low
undeclared.low:13:0 E-ERR-UNDECLARED: op returns an error not in its errors clause
```

`too_long` is a variant of `parse_error`, but it is not in `first_byte`’s `errors` clause. A caller reading the head would believe handling `empty` is enough. When what is written and what is produced differ, one side meets failures it never saw and the other handles failures that never come.

11.3 `try` — passing failure upwards

Writing the failure check by hand every time makes code long, and long code skips checks. `try` takes a `result` and, on success, takes out the value; on failure it **returns that error as is and leaves the op**. `halve_plus_one` in `halve.low` has that shape.

```
let v u8 be try halve a .
return ok (add v 1) .
```

Notice that `halve_plus_one` writes `errors` in its own head too. To pass an error up with `try`, it must itself be able to return that error, and it must say so in its contract. There is no path by which a failure silently disappears. To handle the failure here instead of passing it up, ask with `is_error` and take out with `ok_value`, as `halve_or_zero` does.

> **A common misconception. `try` is the `try` of exception handling**
>
> Java's or C++'s `try` is a place that catches exceptions thrown anywhere in a block. Lowent's `try` is attached to **one expression** and marks that expression's failure as **passed upwards**; it is closer to Rust's `?`. There is no throw-and-catch control flow. Failures always come back as values, and where they can be passed on is written in the source.

11.4 Crossing between the two channels

Sometimes the calling op and the called op use different channels. A tail on `try` changes the container.

examples/ch11/tails.low

```
module tails .
rem run: maybe_half 100
rem run: maybe_half 250
rem run: must_find 2
rem run: must_find 9

enum lookup_error do
  too_big .
  not_found .
end

fn halve input a u8 . output result u8 lookup_error .
  errors too_big gt a 200 .
do
  guard le a 200 . else return error too_big .
  return ok (div a 2) .
end

fn find input k u8 . output option u8 .
do
  guard lt k 3 . else return none .
  return some (mul k 10) .
end

fn maybe_half input a u8 . output option u8 .
do
```

```
  return try (halve a) else_none .
end

fn must_find input k u8 . output result u8 lookup_error .
  errors not_found .
do
  return try (find k) else_error not_found .
end
```

Output

```
$ lowentc --run maybe_half tails.low 100
maybe_half(100) = some 50
$ lowentc --run maybe_half tails.low 250
maybe_half(250) = none
$ lowentc --run must_find tails.low 2
must_find(2) = ok 20
$ lowentc --run must_find tails.low 9
must_find(9) = err not_found
```

Shape	Direction	What is lost or gained
`try <expr> else_none`	`result` → `option`	The error is **discarded**; why it failed is no longer said
`try <expr> else_error <variant>`	`option` → `result`	Absence **gets a name**

Table 11.1 — Changing channel with a `try` tail

A `try` with a tail also changes the type. The type of `try (halve a) else_none` is `option u8`, not `u8`. That is why `maybe_half` returns it as is, and why putting it into a value type, as in `let v u8 be try … else_error …`, is rejected.

`else_none` is a choice that throws information away. It is convenient, so it easily becomes a habit, but from that moment the caller can no longer ask “why”. Throw it away only where it is worth throwing away.

11.5 Three ways to report failure

What	What it says	What the caller does
`result t e`	A failure that can be fixed	Asks which it is and handles it, or passes it up
`option t`	There is no value	Asks whether it exists and takes it, or gives a fallback
`panic` · contract violation	A promise was broken	Cannot handle it; the program stops

Table 11.2 — Three ways to report failure

The question that separates the three is “what can the caller do?”. If a file is missing, another file can be tried, so it is a `result`. If what you look for is not in the list, it simply is not there, so it is an `option`. If the caller broke a contract, the promise is already broken and cannot be fixed, so execution stops. `panic` does not unwind; there is no way to catch it midway and carry on. How to design an op’s failures on this criterion is revisited in chapter 17.

11.6 Combining and nesting patterns

Now that `option`, `result` and `enum` have all appeared, `match` patterns can be used more widely.

examples/ch11/patterns.low

```
module patterns .
rem run: warm 1
rem run: combine 1
rem run: pick 1
rem run: choose
rem run: half 200

enum color do
  red .
  green .
  blue .
end

enum node do
  lit v u32 .
  plus l u32 r u32 .
  times l u32 r u32 .
end

rem any one match takes the arm --- red, green and blue are all covered, so no _ is need
ed
fn warm input k u8 . output u8 . do
  let c color be green .
  match c do
    case red or green . do return 1 . end
    case blue . do return 2 . end
  end
end

rem arms joined with or must bind the same names
fn combine input k u8 . output u32 . do
  let e node be node.times 6 7 .
  match e do
    case plus l r or times l r . do return add l r . end
    case lit v . do return v . end
  end
end

rem splits an option inside a result in one go
fn pick input k u8 . output u32 . do
  let r result option u32 color be ok (some 42) .
  match r do
    case ok (some x) . do return x . end
    case ok none . do return 0 . end
    case error . do return 99 . end
  end
end

rem if the value being split is a compile-time constant, it folds to the one matching ar
m
fn choose output u32 . do
```

```
  match comptime (add 2 3) do
    case 0 . do return 100 . end
    case 5 . do return 500 . end
    case _ . do return 999 . end
  end
end

rem the two ranges cover all of u8 without gaps, so no _ is needed
fn half input b u8 . output u8 . do
  match b do
    case 0 to 127 . do return 0 . end
    case 128 to 255 . do return 1 . end
  end
end
```

Output

```
$ lowentc --run warm patterns.low 1
warm(1) = 1
$ lowentc --run combine patterns.low 1
combine(1) = 13
$ lowentc --run pick patterns.low 1
pick(1) = 42
$ lowentc --run choose patterns.low
choose() = 500
$ lowentc --run half patterns.low 200
half(200) = 1
```

- **Or-patterns.** `case red or green .` is taken if either matches. Each alternative counts towards exhaustiveness, so once `blue` is handled no _ is needed. When or-ing variants that carry values, **every alternative must bind the same names** — `combine` takes out `l` and `r` whether the variant is `plus` or `times`. Mismatched names are `E-MATCH-ORBIND`.
- **Nested patterns.** `case ok (some x) .` splits the `option` inside a `result` in one go. Nested patterns short-circuit, so if it is not `ok` the inner part is never looked at. That is why `error` never tries to take out a value and stop.
- **Folded at translation time.** If the value being split is a translation-time constant — a literal, `comptime <expr>`, `config <name>` — the `match` folds to the one matching arm, with no comparison at run time. Dead arms are still type-checked. That is the difference from C's `#ifdef` (chapter 31).
- **Ranges cover a type.** The two ranges in `half` cover 0 … 255 of `u8` without a gap, so it is exhaustive without _. A gap is `E-MATCH-INEXHAUSTIVE`; overlapping ranges are `E-MATCH-REDUNDANT`.

An arm after _ is rejected.

examples/ch11/arm_after_wild.low

```
module arm_after_wild .
rem expect: E-MATCH-REDUNDANT

enum light do
  lit .
  dark .
end

fn name input k u8 . output u8 . do
```

```
  let s light be lit .
  match s do
    case _ . do return 0 . end
    case lit . do return 1 . end
  end
end
```

Output

```
$ lowentc --check arm_after_wild.low
arm_after_wild.low:13:0 E-MATCH-REDUNDANT: this `case` comes AFTER a `_` (wildcard) arm
— the wildcard already matched, so this arm can never run. A dead arm is an error here,
not a warning (RFC-0020 §6.4)
```

_ has already taken everything, so the arm after it can never run. A dead arm is an error, not a warning — a `match` where each case does not appear exactly once when read hides defects.

11.7 Common mistakes

Counter-example. Using an `option` in arithmetic as if it were a number

examples/ch11/mistake_optarith.low

```
module mistake_optarith .
rem expect: E-TYPE-RETURN

fn find input k u64 . output option u64 .
do
  guard lt k 5 . else return none .
  return some (mul k 10) .
end

fn plus_one input k u64 . output u64 .
do
  rem ✘ an `option u64` is not a number --- ask whether it is there and take it out befo
re adding
  return add (find k) 1 .
end
```

Output

```
$ lowentc --check mistake_optarith.low
mistake_optarith.low:13:0 E-TYPE-RETURN: the returned value does not match the op's `out
put` — expected `u64`, found `option …`
```

`find k` does not give back a `u64`; it gives back “a box that may or may not hold a `u64`”. You cannot add 1 to a box. In other languages a null flows into the calculation and blows up much later; Lowent stops you right here with `E-TYPE-RETURN`. There are three fixes: supply a stand-in with `value_or (find k) 0`, ask with `is_some` and take it out with `some_value`, or split with `match`. Which one to pick depends on “what should happen when it is absent”.

Counter-example. Forgetting `some` in an op that returns an `option`

examples/ch11/mistake_nosome.low

```
module mistake_nosome .
rem expect: E-TYPE-RETURN

fn find input k u64 . output option u64 .
do
  guard lt k 5 . else return none .
  rem ✘ the value is not wrapped in `some`
  return mul k 10 .
end
```

Output

```
$ lowentc --check mistake_nosome.low
mistake_nosome.low:8:0 E-TYPE-RETURN: the returned value does not match the op's `output
` — expected `option …`, found `u64`
```

Once the head says `output option u64`, the value you return must be a box too. `none` is a box, but `mul k 10` is a bare number, so this is `E-TYPE-RETURN`. Some languages wrap the value for you; Lowent does not. Writing `return some (mul k 10) .` spells out “it is there”, so the reader sees both branches.

Counter-example. Using `try` in an op that does not return a `result`

examples/ch11/mistake_trynoresult.low

```
module mistake_trynoresult .
rem expect: E-TRY-NORESULT

enum half_error do
  too_big .
end

fn halve input a u8 . output result u8 half_error .
errors too_big gt a 200 .
do
  guard le a 200 . else return error too_big .
  return ok (div a 2) .
end

rem ✘ uses `try`, but this op neither returns a `result` nor has an `errors` clause
fn plus input a u8 . output u8 .
do
  let v u8 be try halve a .
  return add v 1 .
end
```

Output

```
$ lowentc --check mistake_trynoresult.low
mistake_trynoresult.low:18:0 E-TRY-NORESULT: `try` may only be used in an op that can RE
TURN that error (§6.5.8(2)): this op's output is not a `result`, so there is nowhere for
```

```
 the failure to go. It used to pass, and the VM then returned the ERROR VALUE in the pla
in output slot while the native build would not compile at all. Declare `output result <
T> <E> .` (and the matching `errors` clause), or handle the failure here (`guard is_ok …
` / `value_or` / a `case error` arm)
```

`try` **passes failure upwards**, so this op must be able to return that failure itself (§6.5.8(2)). `plus` returns only a `u8` and has no `errors` clause, so it is refused with `E-TRY-NORESULT`. Until 2026-09-16 it passed: `plus 250` printed `err too_big` where a `u8` belongs, and the native build would not even compile. Adding a tail (`else_none`, `else_error`) or handling it here (`is_ok` and the rest of §6.5.8(3)) keeps the op free of `result`; to pass the failure up, give the head a `result` output and an `errors` clause.

examples/ch11/trynoresult_fixed.low

```
module trynoresult_fixed .
rem run: plus 10
rem run: plus 250

enum half_error do
  too_big .
end

fn halve input a u8 . output result u8 half_error .
errors too_big gt a 200 .
do
  guard le a 200 . else return error too_big .
  return ok (div a 2) .
end

rem an op that passes failure upward returns a result itself and lists the errors it may
 pass in `errors`
fn plus input a u8 . output result u8 half_error .
errors too_big gt a 200 .
do
  let v u8 be try halve a .
  return ok (add v 1) .
end
```

Output

```
$ lowentc --run plus trynoresult_fixed.low 10
plus(10) = ok 6
$ lowentc --run plus trynoresult_fixed.low 250
plus(250) = err too_big
```

Nested patterns count towards exhaustiveness too. An outer tag is covered when **its inner pattern is itself exhaustive**.

examples/ch11/nestedwild.low

```
module nestedwild .
rem run: via_match

enum read_error do
```

```
  broken .
end

fn pick input r result (option u64) read_error . output u64 .
do
  rem three nested patterns are every case --- no separate `_`
  match r do
    case ok (some x) . return x .
    case ok none . return 0 .
    case error e . return 1 .
  end
end

fn via_match output u64 .
do
  return pick (ok (some 42)) .
end
```

Output

```
$ lowentc --run via_match nestedwild.low
via_match() = 42
```

`ok (some x)` and `ok none` together cover `ok`, and `error e` covers the rest, so no _ is needed. Leave one case out and it is refused with `E-MATCH-INEXHAUSTIVE` — better than a _ that covers nothing. A _ says nothing when a variant is added later.

A common misconception. With `value_or` you can still tell when a value was absent

examples/ch11/valueor_blind.low

```
module valueor_blind .
rem run: find_or_zero 0
rem run: find_or_zero 7
rem run: found 0
rem run: found 7

fn find input k u8 . output option u8 .
do
  guard lt k 3 . else return none .
  return some (mul k 10) .
end

rem slot 0 really holds 0 and slot 7 is absent --- yet both come out as 0
fn find_or_zero input k u8 . output u8 .
do
  return value_or (find k) 0 .
end

rem if you need to know whether it is there, ask before covering it with a stand-in
fn found input k u8 . output bool .
do
  return is_some (find k) .
end
```

Output

```
$ lowentc --run find_or_zero valueor_blind.low 0
find_or_zero(0) = 0
$ lowentc --run find_or_zero valueor_blind.low 7
find_or_zero(7) = 0
$ lowentc --run found valueor_blind.low 0
found(0) = 1
$ lowentc --run found valueor_blind.low 7
found(7) = 0
```

`value_or` **covers** absence with a stand-in. When the stand-in collides with a real value, the two cannot be told apart. Above, the real 0 in slot 0 and the absence in slot 7 both come out as 0. Use `value_or` only where "treat absent as 0" is truly fine; when presence matters, ask with `is_some` or `match` before covering it.

11.8 This chapter's syntax at a glance

Shape	Meaning	Why
`output option u8 .`	a value, or none	absence (null) shows in the type
`some v · none`	present · absent	"present" is written too, so both branches are visible
`output result u8 e .`	a value or an error (a variant of `e`)	failure is returned as a value — there are no exceptions
`ok v · error too_big`	success · failure	which branch is written in the source
`errors too_big <condition> .`	promise which error happens when	written in the contract so the caller can prepare
`is_some r · some_value r`	ask whether present · take it out	taking out is partial — ask first
`is_error r · ok_value r`	ask whether failed · take out the success value	same reason
`value_or r 99`	a stand-in when absent	one line, but it covers absence
`try <expr>`	on failure, leave returning that error	so checks are never forgotten — like Rust's `?`
`try <expr> else_none · else_error e`	`result` → `option` · `option` → `result`	changing channel shows what is lost
`case ok (some x) . · case a or b .`	nested pattern · several variants at once	split in one go, still covering every case

Table 11.3 — Syntax of answer-carrying types — shape · meaning · why it looks this way

Recap

`option` is made with `some` and `none`, `result` with `ok` and `error`. The receiver asks and takes out (`is_some`, `some_value`, `is_error`, `ok_value`), gives a fallback with `value_or`, or splits with `match`. Taking out is partial, so taking from the missing side stops. An op returning a `result` promises its errors in an `errors` clause, `try` passes failures up, and the `else_none` and `else_error` tails change channel and type. Patterns can be or-ed (binding the same names) and nested, and a `match` on a constant folds at translation.

12 Borrowing — `ref` and `mut_ref`

In this chapter

12.1 Two kinds of borrow 145
12.2 Write permission only narrows 146
12.3 Many readers or one writer 147
12.4 A borrow cannot outlive what it borrows 148
12.5 There is no `mut ref slice` 149
12.6 Common mistakes 150
12.7 This chapter's syntax at a glance 153

What to know first

chapter 9, Sequences · elements can be written only through a `mut slice`
chapter 10, Aggregates · `field` is a place to read and to write

Looking back

In chapter 9, what rejected writing to an element of a slice without `mut`? And how does immutability of a name made with `let` differ from immutability of a slice's elements?

A. It was rejected with `E-TYPE-MUT`. `let` forbids **putting a different value into the name**, while whether elements can be written is decided by the slice's type (`slice` or `mut slice`). This chapter covers references that borrow a single value that is not a slice, and the rules for when borrows overlap.

The need for this chapter, and its context

Copying a big value every time you pass it is slow, and passing a pointer leaves you unable to tell who changes the value and when. Lowent puts **borrowing** between the two. Borrowers for reading may be many; a borrower for writing must be alone. That rule is the foundation of memory safety without a garbage collector, and the proof that there are no data races in concurrency (Part VII) stands on it too. Before memory (Part V), learn the shape of borrowing.

> ### By the end of this chapter
>
> You will learn to borrow values with `ref t` and `mut_ref t` and read them with `deref`. You will see that a read borrow cannot write and that write permission only narrows in one direction. You will meet the exclusivity rule — borrows of one value are “many readers or one writer” — and the rule that a reference to a local cannot leave its op. You will also understand why `mut ref slice` is rejected.

> ### The questions this chapter answers
>
> 1 How do you make a null reference?

12.1 Two kinds of borrow

A **reference** points to a value that lives elsewhere. There are only two kinds, and the name is the permission.

- `ref t` — a read borrow. Many can hold one at the same time. It cannot change the value.
- `mut_ref t` — a write borrow. There can be only one for a value at a time, and no read borrows meanwhile.

Think of a library book. Many people can **read** it at once, but to **write corrections** in it you must borrow it alone, and nobody reads it meanwhile — otherwise a reader would see half-made corrections.

examples/ch12/borrow.low

```
module borrow .
rem run: use_sum 7
rem run: use_bump 7

struct big do
  a u64 .
  b u64 .
  c u64 .
  d u64 .
end

fn total input v ref big . output u64 .
  requires le (field v a) 1000 .
  requires le (field v b) 1000 .
  requires le (field v c) 1000 .
  requires le (field v d) 1000 .
do
  return add (add (field v a) (field v b)) (add (field v c) (field v d)) .
end

fn use_sum input x u64 . output u64 .
  requires le x 1000 .
do
  let v big be make big do a x . b 2 . c 3 . d 4 . end .
  return add (total (ref v)) (total (ref v)) .
end
```

```
proc bump input p mut_ref u64 . output u64 .
do
  set p (add (deref p) 1) .
  return deref p .
end

proc use_bump input start u64 . output u64 .
  requires lt start 1000 .
do
  var n u64 be start .
  let r u64 be bump (mut_ref n) .
  return add n r .
end
```

Output

```
$ lowentc --run use_sum borrow.low 7
use_sum(7) = 32
$ lowentc --run use_bump borrow.low 7
use_bump(7) = 16
```

- `total` borrows the four-field `big` with `ref big` to read it without copying. `field v a` reads a field through the reference.
- `use_sum` borrows the same value twice with `ref v`. Read borrows may be many.
- `bump` takes a `mut_ref u64` and changes the caller's value with `set p …`. The value a reference points to is read with `deref p`.
- `use_bump` lends `var n` as `mut_ref n`. After `bump` returns, `n` is 8.

What matters is that the borrower **writes `ref x` or `mut_ref n` at the call site**. Someone reading the call sees from that place alone that this call can change `n`.

> **Q. How do you make a null reference?**
>
> **A.** You cannot. A reference always points to a live value. To express "points to nothing", use an `option` (chapter 11). Then the code that checks before taking out shows in the source, and taking out without checking stops.

12.2 Write permission only narrows

Writing through a read borrow is rejected.

examples/ch12/ref_write.low

```
module ref_write .
rem expect: E-TYPE-REF

proc reset input p ref u64 . output u64 .
do
  set p 0 .
  return 0 .
end
```

Output

```
$ lowentc --check ref_write.low
ref_write.low:6:0 E-TYPE-REF: write through a shared ref (declare mut_ref)
```

The other direction is blocked too. Passing a value received only for reading to a position that takes `mut`, `mut_ref` or `owned` is rejected with `E-TYPE-ARGMUT`. Something writable can be passed on for reading, but something read-only cannot be passed to a writable position. Whoever received a value for reading must be able to trust that **it does not change while they look at it**, and that trust holds only when nobody gains write permission behind their back.

12.3 Many readers or one writer

The trouble is overlapping borrows of the same value. Two overlapping write borrows are rejected.

examples/ch12/excl.low

```
module excl .
rem expect: E-EXCL

proc set_both input a mut_ref u64 . input b mut_ref u64 . output void .
do
  set a 1 .
  set b 2 .
end

proc confused output u64 .
do
  var n u64 be 0 .
  set_both (mut_ref n) (mut_ref n) .
  return n .
end
```

Output

```
$ lowentc --check excl.low
13:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
```

The body of `set_both` believes `a` and `b` are different values. Lending the same `n` twice breaks that belief, and whether 1 or 2 remains depends on the order of writes. Overlapping read and write is rejected too.

examples/ch12/stale.low

```
module stale .
rem expect: E-EXCL

proc look_then_write output u64 .
do
  var n u64 be 0 .
  let r ref u64 be ref n .
  set n 5 .
```

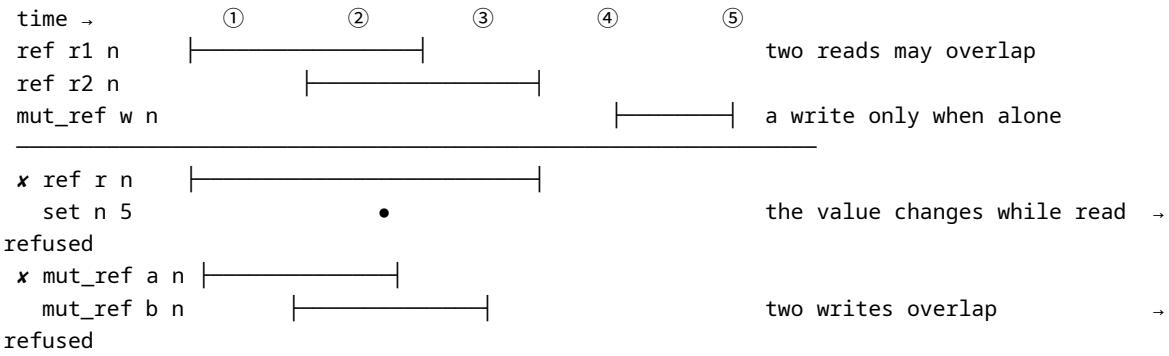
```
  return deref r .
end
```

Output

```
$ lowentc --check stale.low
8:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
```

Writing 5 to `n` while `r` holds a read borrow of `n` means whoever reads through `r` cannot tell when the value they saw changed. These two shapes — another flow changing a value at the same time, and a value changing while being read — are defects hard to find by reading the source, so a rule removes them.

Laying the borrows of one value `n` out in time makes the rule visible at a glance.



> **A common misconception. The exclusivity rule only matters for multithreaded programs**
>
> The two examples above contain no threads. Even within one flow, when two names point at the same value, a write through one breaks what the other believes. That is also why C's `memcpy` is undefined for overlapping buffers. But once the rule holds within one flow, the absence of data races when the work is split into several flows follows, and that is where the proofs of Part VII lean (chapter 45).

12.4 A borrow cannot outlive what it borrows

Returning a reference to a local from its op is rejected.

examples/ch12/escape.low

```
module escape .
rem expect: E-ESCAPE

fn leak output ref u32 .
do
  let here u32 be 42 .
  return ref here .
end
```

Output

```
$ lowentc --check escape.low
7:0 E-ESCAPE: reference to a local escapes the op (dangling)
```

If this code were translated, the returned reference would point at a value already gone, and nobody knows what reading it would give. Lowent knows this not by running it but **at translation time**. A borrow cannot leave the block that opened it (`E-BORROW-ESCAPE`), and a borrow still alive after the lent value has moved is rejected too (`E-EXCL-MOVED`).

If a value must leave the op, return **the value** rather than a reference, or put it in storage the caller passed. If you need storage that outlives the caller, use a region (chapter 18).

12.5 There is no `mut ref slice`

Slices do not take `mut ref`.

examples/ch12/mref_slice.low

```
module mref_slice .
rem expect: E-MREF-SLICE

proc shrink input p mut ref slice u8 . output u64 .
do
  return 0 .
end
```

Output

```
$ lowentc --check mref_slice.low
mref_slice.low:4:0 E-MREF-SLICE: `mut ref slice` gives nothing that `mut slice` does not
 — the elements are already writable through a plain `mut slice` (the descriptor is copi
ed, the bytes are shared). The ONLY thing it adds is replacing the caller's descriptor,
which silently changes the length behind the caller's back. Say the new slice with a RET
URN VALUE instead (SPEC-004 §4.4a: the windows through which someone else can change you
r local are listed, and this one is closed)
```

Passing a `mut slice` already makes the elements writable. The slice value `{start, length}` is copied, but the bytes it points to are the same. So what `mut ref slice` adds is exactly one thing — **secretly changing the length and start of the caller's slice**. Then the caller's loop has its length change even though its own code contains no assignment. Return the new slice instead.

examples/ch12/head.low

```
module head .
rem run: first_two [9,8,7,6]

fn take_front input p slice u8 . input n u64 . output slice u8 .
  requires le n (len p) .
do
  return subslice p 0 n .
end

fn first_two input xs slice u8 . output u64 .
  requires ge (len xs) 2 .
do
```

```
  let front slice u8 be take_front xs 2 .
  return len front .
end
```

Output

```
$ lowentc --run first_two head.low [9,8,7,6]
first_two([9,8,7,6]) = 2
  arg0 (written) = [9,8,7,6]
```

`take_front` returns a new slice pointing at the first `n` elements. That the length changed shows in the return value and in `let front`.

12.6 Common mistakes

Counter-example. Forgetting `mut_ref` at the call site

examples/ch12/mistake_nomutref.low

```
module mistake_nomutref .
rem expect: E-TYPE-ARG

proc bump input p mut_ref u64 . output u64 .
do
  set p (add (deref p) 1) .
  return deref p .
end

proc use_bump input start u64 . output u64 .
  requires lt start 1000 .
do
  var n u64 be start .
  rem ✘ `mut_ref` is missing at the call --- the number 7 is passed, not a reference
  let r u64 be bump n .
  return add n r .
end
```

Output

```
$ lowentc --check mistake_nomutref.low
mistake_nomutref.low:15:0 E-TYPE-ARG: this op takes a BORROW here (`ref` / `mut_ref`), a
nd a plain value was passed. Take the borrow at the call — `mut_ref <name>` — so the rea
der sees where the callee may write. It used to pass `--check` and stop at run time with
 `E-VM-TYPE: deref needs a reference`
```

`bump` expects “a reference to a number”, but `bump n` passes the number 7 itself. C++ reference parameters need no mark at the call, but Lowent makes you write `mut_ref n` so the call alone tells you what may change. This edition’s tool does not reject the omission at translation time; at run time `deref` meets something that is not a reference and stops (recorded as a defect in the development repository). When the stop says “deref needs a reference”, look for a missing `mut_ref` or `ref` at the call site.

Counter-example. Using a reference as a number without `deref`

examples/ch12/mistake_noderef.low

```
module mistake_noderef .
rem expect: E-TYPE-REFVAL

fn twice input p ref u64 . output u64 .
do
  rem ✘ `p` is not a number but a reference to one --- read it with `deref p`
  return add p p .
end

fn double input x u64 . output u64 .
  requires le x 1000 .
do
  let n u64 be x .
  return twice (ref n) .
end
```

Output

```
$ lowentc --check mistake_noderef.low
mistake_noderef.low:7:0 E-TYPE-REFVAL: a BORROW is being used where a number is expected
. `ref t` / `mut_ref t` names a place, not the value in it — read it with `deref <name>`
 (and write through it with `set <name> …`). It used to pass `--check` and stop at run t
ime with `E-VM-TYPE`, which blamed the arithmetic instead of the missing read
```

A reference is **where** a value lives, not the value. Each read spells `deref p`, making "read the borrowed value here" visible. In C, dropping the `*` turns the code into address arithmetic; Lowent has no address arithmetic, so a wrong value never comes out quietly. This edition's tool does not report it at translation time, though, and stops at run time instead (recorded as a defect). The fixed line is `add (deref p) (deref p)`.

Counter-example. Lending a `let` name for writing

examples/ch12/mistake_letmutref.low

```
module mistake_letmutref .
rem expect: E-TYPE-ARGMUT

proc bump input p mut_ref u64 . output void .
do
  set p (add (deref p) 1) .
end

proc use_bump input start u64 . output u64 .
  requires lt start 1000 .
do
  rem ✘ a name made with `let` is lent for writing
  let n u64 be start .
  bump (mut_ref n) .
  return n .
end
```

Output

```
$ lowentc --check mistake_letmutref.low
mistake_letmutref.low:14:0 E-TYPE-ARGMUT: a WRITE borrow (`mut_ref`) was taken of a name
 that cannot be written: a `let` binding (or a shared input). `let` says the value does
not change (§6.5.1) — if a borrow could change it, the reader who checked that name once
 would be wrong, and the borrow makes the change invisible at the call site. Bind it wit
h `var`, or take a read borrow (`ref`)
```

`let` promises "the value of this name does not change". Lending such a name with `mut_ref` breaks that promise, so it is refused with `E-TYPE-ARGMUT`. A borrow changes the value invisibly at the call site, which is exactly what would cost `let` its worth: that checking the name once is enough. Make a value that must change a `var` from the start, so the reader learns from the declaration that it changes somewhere.

A common misconception. A variable passed as `input` can be changed by the op it is passed to

examples/ch12/copyparam.low

```
module copyparam .
rem run: use_inc 7

proc inc input x u64 . output u64 .
  requires lt x 1000 .
do
  rem `x` is a copy of the caller's value --- nothing done here reaches the caller's var
iable
  var y u64 be x .
  set y (add y 1) .
  return y .
end

proc use_inc input start u64 . output u64 .
  requires lt start 1000 .
do
  var n u64 be start .
  let r u64 be inc n .
  rem `n` is still 7 --- lend it with `mut_ref n` to let it change
  return n .
end
```

Output

```
$ lowentc --run use_inc copyparam.low 7
use_inc(7) = 7
```

A plain value parameter (`input x u64`) receives a **copy** of the caller's value. Whatever `inc` does inside, `n` in `use_inc` stays

7. Some languages pass large values by reference behind your back and blur this line; in Lowent the only way to change a caller's

value is `mut_ref`, and that mark stays at the call site.

> **A common misconception. Within one op you may lend a value with `mut_ref` only once**

examples/ch12/seqborrow.low

```
module seqborrow .
rem run: use_twice 7

proc bump input p mut_ref u64 . output void .
  requires lt (deref p) 1000 .
do
  set p (add (deref p) 1) .
end

proc use_twice input start u64 . output u64 .
  requires lt start 900 .
do
  var n u64 be start .
  rem the first borrow comes back when this call ends --- so the next line may borrow ag
ain
  bump (mut_ref n) .
  bump (mut_ref n) .
  return n .
end
```

Output

```
$ lowentc --run use_twice seqborrow.low 7
use_twice(7) = 9
```

The exclusivity rule forbids borrows that **overlap in time**. A `mut_ref n` passed to one call comes back when that call ends, so the next line may borrow again. What is rejected is a shape like `set_both (mut_ref n) (mut_ref n)`, where two borrows live inside a single call.

12.7 This chapter's syntax at a glance

Shape	Meaning	Why
`input v ref big .`	receive a read borrow	no copy of a large value, and a promise not to change it
`input p mut_ref u64 .`	receive a write borrow	the only way to change a caller's value
`total (ref v) • bump (mut_ref n)`	lend at the call site	the call alone shows what may change
`deref p`	read the value a reference points to	a value and its place are never mixed up
`set p <value> .`	write where a reference points (`mut_ref`)	through a read borrow it is `E-TYPE-REF`
`field v a`	read a field through a reference	same spelling for a reference to a struct

two `mut_ref` of one value · `ref` plus a write	overlapping borrows — rejected (`E-EXCL`)	many readers or one writer
one storage in a written and a read position	only when the callee allows it with `inplace <written> <read> .` and the ranges are **the same** — else `E-EXCL-INPLACE`	a body that reads and writes element by element is right only on the same range
the body of an op that declares `inplace o a`	write `o` only after every read of `a`, or read and write element by element with one index — else `E-INPLACE-UNPROVEN`	the processor checks the declaration against the body
using an old view after calling an op that declares `invalidates <input>` on its storage	rejected (`E-VIEW-INVALIDATED`) — take the view again	after a release, a growth or a rewind the view points at someone else's place
`return ref here .`	return a reference to a local — rejected	never point at a value that is gone
`mut ref slice`	does not exist — rejected (`E-MREF-SLICE`)	no hidden length change — return a new slice

Table 12.1 — Borrowing syntax — shape · meaning · why it looks this way

Recap

`ref t` is a read borrow and `mut_ref t` a write borrow, and the borrower writes `ref x` or `mut_ref n` at the call site. The value a reference points to is read with `deref`. Write permission only narrows. Borrows of one value are many readers or one writer, and a borrow cannot outlive the lent value. `mut ref slice` is rejected; return the new slice.

13 Named types — `type`, `newtype`, `range`, `cast`

In this chapter

13.1 `type` is an alias, `newtype` a new type 155
13.2 `range` — a contract that became the shape of a parameter 158
13.3 `cast` — where a value may change 159
13.4 `bits` — from 1 to 64 bits 160
13.5 Pinning down byte layout 160
13.6 Both directions of a layout, and a view that can fail 161
13.7 `bitset` — a set of small numbers 162
13.8 Scattered pieces as one view — `segments` 164
13.9 Three kinds of type words 165
13.10 Common mistakes 166
13.11 This chapter's syntax at a glance 169

What to know first

chapter 4, Numbers · widening is automatic, narrowing is written
chapter 10, Aggregates · `struct` and `field`

Looking back

Chapter 4 said `usize` and `u64` do not convert automatically even on machines where they have the same width. Why?

> **A.** To make places mixing address counts with plain numbers visible in the source. When the meaning differs, the type differs even if the representation is the same, and crossing over has to be written. This chapter covers how users apply the same principle themselves — `newtype` — and how to write ranges, widths and layouts into types.

The need for this chapter, and its context

If user numbers and order numbers are both `u64`, swapping them goes unnoticed by the compiler. So does passing 300 to a parameter meant for a percentage. These defects arise because a type says only its **representation** and not its **meaning**. As the last chapter of Part III, it gathers the ways to attach names, ranges and layouts to types. It sits right before contracts (Part IV) because `range` has the same standing as a contract.

By the end of this chapter

You will learn that `type` is another name for the same type and `newtype` a different type with the same representation, and that `cast` crosses between them. You will pick up what `range lo hi` in a parameter position gives the caller and the body, and how `cast` differs from `widen` and `narrow`. You will also see `bits` integers from 1 to 64 bits, and `layout packed` and `view`, which pin down byte layout.

The questions this chapter answers

1 Does a `newtype` cost anything at run time?

13.1 `type` is an alias, `newtype` a new type

`type <name> <type> .` gives an existing type another name. The two names are the same type and interchangeable.

examples/ch13/aliases.low

```
module aliases .
rem run: area 3 4
rem run: half_percent 90

type meters u64 .
type pct range 0 100 .

fn area input w meters . input h meters . output u64 .
```

```
  requires le w 1000000 .
  requires le h 1000000 .
do
  return mul w h .
end

fn half_percent input p pct . output u8 .
do
  return narrow u8 (div p 2) .
end
```

Output

```
$ lowentc --run area aliases.low 3 4
area(3, 4) = 12
$ lowentc --run half_percent aliases.low 90
half_percent(90) = 45
```

`meters` is just another name for `u64`, so the result of `mul w h` can be returned as a `u64`. Aliases shorten long types (`type bytes slice u8 .`) or leave meaning in the source. `pct` is an alias for a type with a range attached, and every place that takes this name inherits the range (covered below).

A type declaration has no `be`.

examples/ch13/type_be.low

```
module type_be .
rem expect: E-TYPE-DECL

type pct be u8 .
```

Output

```
$ lowentc --check type_be.low
type_be.low:4:0 E-TYPE-DECL: a type alias is `type N T .` and a newtype is `newtype N T
.` — without `be`. `be` binds a VALUE (`let x u8 be 1 .`); a type declaration names a TY
PE. One meaning, one spelling
```

`be` is the word with which `let` and `var` bind **values**. What is bound here is a type. One meaning, one spelling.

`newtype <name> <type> .` makes a **different type** with the same representation as an existing one.

examples/ch13/ids.low

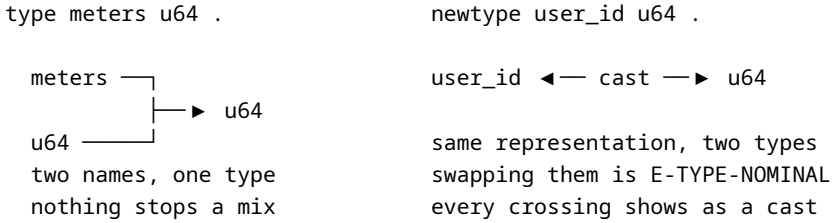
```
module ids .
rem expect: E-TYPE-NOMINAL

newtype user_id u64 .
newtype order_id u64 .

fn load_order input id order_id . output u64 .
do
  return 0 .
end
```

```
fn mistake input u user_id . output u64 .
do
  return load_order u .
end
```

Output

```
$ lowentc --check ids.low
ids.low:14:0 E-TYPE-NOMINAL: these are NOMINALLY DISTINCT types with the same representa
tion — a `newtype` (or usize/isize) is not interchangeable with its base or with another
 newtype. Convert explicitly with `cast <type> <value>` (RFC-0002 §6.3.1)
```

`user_id` and `order_id` are both represented as 64-bit numbers, but cannot be swapped. Such mistakes are hard to find with tests — when the numbers happen to coincide, the wrong order is fetched without a sound. Split them with `newtype` and translation catches it.



When you mean to cross over, write `cast`.

examples/ch13/ids_ok.low

```
module ids_ok .
rem run: roundtrip 42

newtype user_id u64 .

fn user_of input n u64 . output user_id .
do
  return cast user_id n .
end

fn raw_of input u user_id . output u64 .
do
  return cast u64 u .
end

fn roundtrip input n u64 . output u64 .
do
  return raw_of (user_of n) .
end
```

Output

```
$ lowentc --run roundtrip ids_ok.low 42
roundtrip(42) = 42
```

With only one or two crossing ops like `user_of` and `raw_of`, the places where a raw number enters as a `user_id` gather into those two. Those are the only places that need checking.

> **Q. Does a `newtype` cost anything at run time?**
>
> **A.** No. The representation is the same, so in native code `user_id` is just a 64-bit integer. The distinction exists only at translation time, and `cast user_id n` does not change the value. It is a free distinction, worth using freely.

13.2 `range` — a contract that became the shape of a parameter

Writing `range <low> <high>` in a parameter's type position makes that parameter accept only values between the two ends, inclusive.

examples/ch13/ranges.low

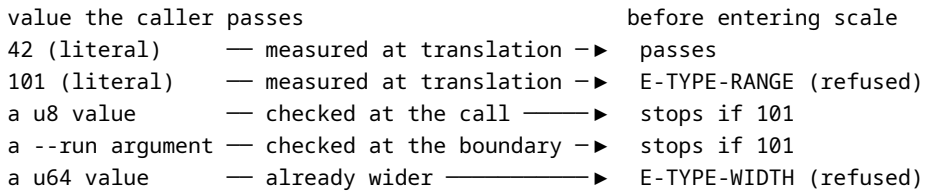
```
module ranges .
rem run: scale 100
rem trap: scale 101

fn scale input a range 0 100 . output u8 .
do
  return narrow u8 (mul a 2) .
end
```

Output

```
$ lowentc --run scale ranges.low 100
scale(100) = 200
$ lowentc --run scale ranges.low 101
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: a parameter's declared range was violated at the program boundary (th
e value entered from outside — no caller proved it)
```

The body of `scale` uses the fact that `a` is at most 100. So `mul a 2` does not exceed 200, `narrow u8` cannot fail, and both overflow checks are removed. Keeping to the range is the caller's responsibility. If the type of the value passed is already wider than the range (such as `u64`), translation is refused; a value of a type that can fit, such as `u8`, is checked at the call. Values that come from **outside** the program (here, `--run`'s arguments) are also checked at the boundary, and 101 stops there. The specification says translation is refused when the caller **cannot prove** the range, but the compiler in this edition hands unproven `u8` values to a run-time check.

Who measures the range, and where, in one picture (the literal `101` was measured on this edition).



The same could be written as `requires le a 100 .`. The difference is **where it is written**. `range` becomes the shape of the parameter, so the caller sees it from the signature alone, and it can be carried into several ops through an alias (`type pct range 0 100 .`).

13.3 `cast` — where a value may change

`widen` is only for places where the value does not change. Conversions that change sign or cross kinds (floating point ↔ integer) are written `cast <type> <value>`.

examples/ch13/casts.low

```
module casts .
rem run: truncate 7.9
rem run: truncate -2.9
rem run: to_signed 200
rem trap: to_signed 3000000000

fn truncate input x f64 . output i32 .
do
  return cast i32 x .
end

fn to_signed input x u32 . output i32 .
do
  return cast i32 x .
end
```

Output

```
$ lowentc --run truncate casts.low 7.9
truncate(7.9) = 7
$ lowentc --run truncate casts.low -2.9
truncate(-2.9) = -2
$ lowentc --run to_signed casts.low 200
to_signed(200) = 200
$ lowentc --run to_signed casts.low 3000000000
== ir diagnostics (1) ==
0:0 E-VM-CAST: value does not fit the target width (use narrow_wrap / narrow_sat / narro
w_try)
```

- Converting floating point to integer truncates towards zero: 7.9 becomes 7, −2.9 becomes −2. Discarding the fraction does not stop.
- If the integer part or the value does not fit the target type, execution stops. Three billion does not fit in an `i32`.
- A `bool` can be neither the target nor the source of a `cast`. To go between booleans and numbers, write an `if`.

`cast` is not a mark meaning “anything goes”. It is a mark meaning **I know the value may change here**. It does not break the principle that no value-losing conversion happens implicitly (chapter 4), because the author wrote the word.

> **A common misconception. `cast` moves the bits unchanged, like a C cast**
>
> C’s `(int32_t)x` truncates or wraps values that do not fit, depending on the implementation. Lowent’s `cast` moves the value and stops when it does not fit. Keeping the bits and changing only how they are read is a separate job, done by `bit_cast` (chapter 20). Two jobs do not share one word.

13.4 `bits` — from 1 to 64 bits

`type <name> bits <count> .` makes an integer type with that many bits. The width need not be a power of two.

examples/ch13/tiny.low

```
module tiny .
rem run: grow 1000
rem trap: grow 1023

type ten_bits bits 10 .

fn grow input x ten_bits . output ten_bits .
do
  return add x 1 .
end
```

Output

```
$ lowentc --run grow tiny.low 1000
grow(1000) = 1001
$ lowentc --run grow tiny.low 1023
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
```

`ten_bits` is 0 … 1023. Adding 1 to 1023 overflows ten bits and stops. When dealing with protocol bit fields or packed tables, the width becomes the contract. But when the width does not match the machine's word, there is no promise of speed. What the language promises is the **correct** answer computed in ten bits.

13.5 Pinning down byte layout

A struct's layout is normally chosen by the processor to suit the machine. When bytes must **mean the same outside** — a header going onto the wire, a register a device reads — write the layout.

examples/ch13/packed.low

```
module packed .

type bytes slice u8 .

struct wire_header do
  layout packed .
  magic u32 big .
  length u16 big .
  kind u8 .
end

fn header_kind input b bytes . output u64 .
do
  let v view wire_header be view wire_header b .
```

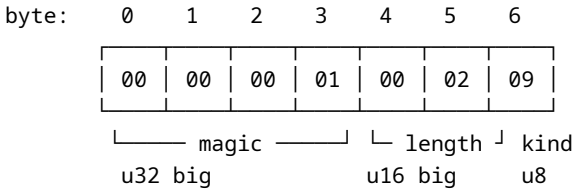
```
  return widen u64 (field v kind) .
end
```

Output

```
$ lowentc --check packed.low
== check: ok ==
```

- `layout packed .` puts no padding between fields. Fields sit next to each other in declaration order.
- `big` and `little` after a field set its byte order. If not written, the machine's order is used.
- `view wire_header b` reads a byte slice as a value of that layout **without copying**. If length or alignment do not fit, execution stops.

A header holding `magic 1 · length 2 · kind 9` lies in bytes like this. `big` puts the high byte first.



A field can also carry an access mark such as `rw`, `ro` or `wo`. In a struct that maps device registers, reading a write-only field is refused at translation (chapter 30). Pinning the layout takes choices away from the processor, so pin it only where needed.

13.6 Both directions of a layout, and a view that can fail

`view` **reads** bytes as a value of that layout. The opposite direction — **making** bytes of that layout from a value — is `encode`.

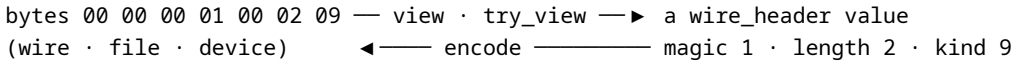
examples/ch13/encoded.low

```
module encoded .
rem run: header_bytes 9

struct wire_header do
  layout packed .
  magic u32 big .
  length u16 big .
  kind u8 .
end

rem the opposite of `view` --- turn a value into bytes of that layout
fn header_bytes input k u8 . output u64 .
do
  let h wire_header be make wire_header do magic 1 . length 2 . kind k . end .
  let bytes slice u8 be encode wire_header h .
  return add (mul (len bytes) 1000) (widen u64 (index bytes 6)) .
end
```

Output

```
$ lowentc --run header_bytes encoded.low 9
header_bytes(9) = 7009
```

`encode wire_header h` produces seven bytes in the byte order written on each field (`big`). The result 7009 puts the length 7 and the 9 of the last byte, `kind`, side by side. The side that builds a header going onto the wire uses `encode`; the side that reads a received header uses `view`.



`view` **stops** when the length or alignment is off, because it treats that as a broken contract. Bytes from a network, however, are often short. In such places use `try_view`, which does the same work but reports failure as a value (chapter 17).

examples/ch13/tryview.low

```
module tryview .
rem run: kind_or_zero [1,2,3,4,0,5,9]
rem run: kind_or_zero [1,2,3]

struct wire_header do
  layout packed .
  magic u32 big .
  length u16 big .
  kind u8 .
end

rem bytes from outside may be short --- `try_view` gives `none` instead of stopping like
 `view`
fn kind_or_zero input b slice u8 . output u64 .
do
  let o option view wire_header be try_view wire_header b .
  guard is_some o . else return 0 .
  let v view wire_header be some_value o .
  return widen u64 (field v kind) .
end
```

Output

```
$ lowentc --run kind_or_zero tryview.low [1,2,3,4,0,5,9]
kind_or_zero([1,2,3,4,0,5,9]) = 9
  arg0 (written) = [1,2,3,4,0,5,9]
$ lowentc --run kind_or_zero tryview.low [1,2,3]
kind_or_zero([1,2,3]) = 0
  arg0 (written) = [1,2,3]
```

With seven bytes it reads `kind` 9; with three it gets `none` and returns 0. Use the stopping `view` inside, where things are already checked, and the value-reporting `try_view` at the boundary.

13.7 `bitset` — a set of small numbers

`bitset <bits>` is a set recording **whether each number** from 0 up to (not including) that count is present. Its name resembles `bits` (the width of one integer), but the meaning differs: `bitset` is

not a tool for the bits of a word but a set, and the bits of a word are handled by bit operations such as `bit_and` and `shl`.

examples/ch13/sets.low

```
module sets .
rem run: overlap 5
rem run: overlap 3
rem run: flipped

rem a collection of which numbers in 0 … 63 are present --- a set, not the bits of a wor
d
fn overlap input n u64 . output u64 .
  requires le n 60 .
do
  var a bitset 64 be bitset_new 64 .
  var b bitset 64 be bitset_new 64 .
  add a 1 .
  add a 3 .
  add a 5 .
  add b 3 .
  add b n .
  let both bitset 64 be intersect a b .
  let only_a bitset 64 be difference a b .
  rem remove 1 from the set --- it changes in place
  remove a 1 .
  var code u64 be add (mul (count both) 100) (mul (count only_a) 10) .
  if is_subset both a . do set code (add code 1) . end
  if is_empty only_a . do set code (add code 1000) . end
  return code .
end

rem a complement only means something within the width --- put one of eight in, and seve
n remain
fn flipped output u64 .
do
  var a bitset 8 be bitset_new 8 .
  add a 0 .
  return count (complement a) .
end
```

Output

```
$ lowentc --run overlap sets.low 5
overlap(5) = 211
$ lowentc --run overlap sets.low 3
overlap(3) = 121
$ lowentc --run flipped sets.low
flipped() = 7
```

- `bitset_new 64` makes an empty set. The width is a number fixed at translation time.
- `add a 1 .` inserts and `remove a 1 .` removes. Both are statements that change the set **in place**. `count a` is the number of members.
- `intersect a b` gives the intersection, `difference a b` what is only in `a`, and `complement a` the complement, each **as a new set**.
- `is_subset x y` asks whether all of `x` is in `y`; `is_empty x` asks whether it is empty.

In `overlap 5` the intersection is {3, 5} and what is only in `a` is {1}, giving 211. `complement` only means something within the width: put 0 into an eight-slot set and its complement is the other seven. Inserting or asking about a number outside the width stops the program — the range of the set is part of its type too.

13.8 Scattered pieces as one view — `segments`

Sometimes bytes do not lie in one row but are scattered over several pieces, like pieces received from the network or the two ends of a ring buffer. Gathering them into one place means copying. `segments` views the pieces as one without copying.

examples/ch13/pieces.low

```
module pieces .
rem run: main

proc main input al cap allocator . output u8 . effects alloc .
do
  let mb option mut slice u8 . . be alloc_bytes al capacity 16 .
  guard is_some mb . else return 250 .
  let md option mut slice u8 . . be alloc_bytes al capacity 32 .
  guard is_some md . else return 251 .
  rem the 16 backing bytes
  var back mut slice u8 . be some_value mb .
  set (index back 4) 20 .
  set (index back 9) 30 .
  rem the descriptor: two (start, length) pairs, 4 from 0 and 4 from 8
  var d mut slice u64 . be view_array u64 (some_value md) .
  set (index d 0) 0 .
  set (index d 1) 4 .
  set (index d 2) 8 .
  set (index d 3) 4 .
  rem see the two pieces as one view, without copying
  let ss segments u8 be view_segments back d .
  let p slice u8 be seg ss 1 .
  rem the piece count 2 times 10, plus byte 1 of the second piece (the 30 at original po
sition 9)
  return narrow u8 (add (mul (segs ss) 10) (index p 1)) .
end
```

Output

```
$ lowentc --run main pieces.low
main() = 50
```

- `view_segments back d` binds the backing bytes `back` and the descriptor `d` into a view of type `segments u8`. The descriptor is a row of (start, length) pairs; here 4 bytes from 0 and 4 bytes from 8.
- `segs ss` gives the piece count 2, and `seg ss 1` gives the second piece as an ordinary `slice u8`. Byte 1 of the second piece is the 30 at original position 9.
- There is no new machine instruction. It lowers to building a grouping, reading fields and slicing, so the cost is visible.

13.9 Three kinds of type words

The words that may stand in a type position form a closed list set by the canon. Some of them are known by name only and have no meaning yet.

examples/ch13/notyet.low

```
module notyet .
rem expect: W-NOT-YET

rem byte is a type word known by name only; it has no meaning yet, so a warning is given
; write a byte as u8
fn first input x byte . output u64 . do
  return 0 .
end

rem `list` in grammar annex A.9 is the same — it was on the list, but no clause ever gav
e it a meaning
fn count input xs list u64 . output u64 .
do
  return 0 .
end
```

Output

```
$ lowentc --check notyet.low
notyet.low:10:0 E-NAME-BUILTIN: this name is a BUILTIN — the resolver always picks the b
uiltin, so your declaration can never be called: it exists and does not exist. The names
pace is FLAT (no shadowing). Rename it
notyet.low:5:0 W-NOT-YET: this type NAME is accepted but carries NO MEANING yet: lowerin
g cannot read it, so every op in its signature falls to the interpreter (~80x). The answ
er stays right, so no oracle will ever see this. Declare it (`type str slice u8 .`) or w
rite the underlying type
notyet.low:10:0 W-NOT-YET: this type NAME is accepted but carries NO MEANING yet: loweri
ng cannot read it, so every op in its signature falls to the interpreter (~80x). The ans
wer stays right, so no oracle will ever see this. Declare it (`type str slice u8 .`) or
write the underlying type
```

`byte` is accepted, but `W-NOT-YET` says “no meaning”. Accepting it silently would make the writer think it works. Write a byte as `u8`.

Kind	Words	What the tool does
usable	numbers · `bool` · `void` · `slice` · `array` · `segments` · `set` · `stack` · `range` · `vec` · `bitset` · `mask` · `result` · `option` · `ref` · `mut_ref` · `mut` · `owned` · `region` · `cap` · `self`	checks and lowers them by meaning
known by name	`byte` · `char` · `str` · `string` · `bytes_view` · `dyn` · `atomic`	`W-NOT-YET` — says there is no meaning
refused for now	`shared_read` · `lock` · `rwlock`	`E-LOCK-NOTYET` — state shared between flows (chapter 26)

qualifier	`unsafe_ptr`	a C pointer mark placed before a type (chapter 29)
no meaning in the canon	`list` · `raw` · `addr` · `rng`	this edition's tool accepts them without a warning — do not use them

Table 13.1 — Type words — usable · known by name · refused

The last row is a hole in this edition. The four words are in the canon's list, but no clause gives them a meaning, and the tool accepts them without a word. It is recorded as a defect in the development repository.

13.10 Common mistakes

Counter-example. Passing a value of a wider type straight to a `range` parameter

examples/ch13/mistake_rangewide.low

```
module mistake_rangewide .
rem expect: E-TYPE-WIDTH

fn scale input a range 0 100 . output u8 .
do
  return narrow u8 (mul a 2) .
end

fn call_scale input x u64 . output u8 .
do
  rem ✘ `u64` is wider than 0 … 100 --- the caller has not shown the value is in range
  return scale x .
end
```

Output

```
$ lowentc --check mistake_rangewide.low
mistake_rangewide.low:12:0 E-TYPE-WIDTH: value does not fit the declared type (a literal
 out of range, or a narrowing — use narrow / narrow_wrap / narrow_sat / round_to)
```

`scale` says in its signature that it takes only 0 … 100. A `u64` value is wider than that, so unless the caller shows it is in range, translation rejects the call with `E-TYPE-WIDTH`. The point of rejecting is to settle who is responsible: only the caller knows what to do with an out-of-range value. Check the range with `guard`, answer separately when it is outside, and pass the value on with `narrow` when it is inside.

examples/ch13/rangewide_fixed.low

```
module rangewide_fixed .
rem run: call_scale 40
rem run: call_scale 400

fn scale input a range 0 100 . output u8 .
do
  return narrow u8 (mul a 2) .
```

```
end

fn call_scale input x u64 . output u8 .
do
  rem answer separately when out of range; otherwise narrow and pass it on
  guard le x 100 . else return 255 .
  return scale (narrow u8 x) .
end
```

Output

```
$ lowentc --run call_scale rangewide_fixed.low 40
call_scale(40) = 80
$ lowentc --run call_scale rangewide_fixed.low 400
call_scale(400) = 255
```

Counter-example. Using `widen` to put a signed number into an unsigned type

examples/ch13/mistake_widensign.low

```
module mistake_widensign .
rem expect: E-WIDEN-SIGN

fn to_count input x i32 . output u64 .
do
  rem ✘ a negative number has no place in `u64` --- `widen` fails even though the width
is larger
  return widen u64 x .
end
```

Output

```
$ lowentc --check mistake_widensign.low
mistake_widensign.low:7:0 E-WIDEN-SIGN: a signed integer cannot widen into an unsigned o
ne — a negative value has no place there. Use `cast` if that is what you mean
```

Going from `i32` to `u64` makes the width larger, but a negative number such as −1 has no place in `u64`. `widen` is only for places where **no value changes**, so this is `E-WIDEN-SIGN`. If you know no negative value can arrive, write `cast u64 x` to leave that judgement in the source; `cast` stops if a negative value does arrive. If negatives need their own handling, put `guard ge x 0 .` first.

Counter-example. Turning a `bool` into a number with `cast`

examples/ch13/mistake_boolcast.low

```
module mistake_boolcast .
rem expect: E-TYPE-KIND

fn flag input b bool . output u8 .
do
  rem ✘ whether true is 1 is for the program to decide, not the language
  return cast u8 b .
end
```

Output

```
$ lowentc --check mistake_boolcast.low
mistake_boolcast.low:7:0 E-TYPE-KIND: bool is not numeric: it cannot be a cast SOURCE ei
ther. Which side is `1` is for the PROGRAM to say, not the language — write `if b . do 1
 . else 0 .`
```

C treats true as 1, but whether true is 1, 0 or −1 is a promise of the program, not something the language should decide. So `bool` can be neither the source nor the target of `cast`, and it is rejected with `E-TYPE-KIND`. As the tool suggests, write “1 if true, 0 if false” with `if`. That one line puts the promise in the source.

A common misconception. Giving a type another name with `type` keeps units from mixing

examples/ch13/type_units.low

```
module type_units .
rem run: total 100 100

type meters u64 .
type feet u64 .

rem meters and feet are added and nothing stops it --- `type` is only another name
fn total input m meters . input f feet . output meters .
  requires le m 1000000 .
  requires le f 1000000 .
do
  return add m f .
end
```

Output

```
$ lowentc --run total type_units.low 100 100
total(100, 100) = 200
```

`meters` and `feet` are both just **other names** for `u64`, so nothing stops you adding them: 100 meters plus 100 feet comes out as 200 meters. Meanings that must not mix are separated with `newtype`; then, as in `ids.low`, translation stops the mix with `E-TYPE-NOMINAL`, and every crossing shows up as a `cast`. Use `type` to shorten long names or to hand down a range.

A common misconception. `cast` of a negative number into an unsigned type gives a big number, as in C

examples/ch13/negcast.low

```
module negcast .
rem run: to_unsigned 5
rem trap: to_unsigned -1

fn to_unsigned input x i32 . output u32 .
do
  rem in C, -1 would become 4294967295 --- Lowent stops when the value does not fit
```

```
  return cast u32 x .
end
```

Output

```
$ lowentc --run to_unsigned negcast.low 5
to_unsigned(5) = 5
$ lowentc --run to_unsigned negcast.low -1
== ir diagnostics (1) ==
0:0 E-VM-CAST: value does not fit the target width (use narrow_wrap / narrow_sat / narro
w_try)
```

In C, `(uint32_t)-1` is 4294967295. That wrap-around is defined by the standard, but it usually hides a bug. Lowent's `cast` moves the value and stops with `E-VM-CAST` when it does not fit. If you really want wrapping, use a word that carries that meaning in its name, such as `narrow_wrap` (chapter 4).

13.11 This chapter's syntax at a glance

Shape	Meaning	Why
`type meters u64 .`	another name for the same type	shortens a type and records meaning — does not stop mixing
`newtype user_id u64 .`	a new type with the same representation	translation stops ids from mixing — no run-time cost
`cast user_id n • cast u64 u`	cross between the new type and the original	crossings gather in one place in the code
`input a range 0 100 .`	accept only values in the range, both ends included	the contract becomes the shape of the signature
`type pct range 0 100 .`	put a range on an alias	many ops inherit the same range
`cast i32 x`	a conversion that may change the value — stops if it does not fit	a mark that says "I know the value may change here"
`type ten_bits bits 10 .`	an integer of 1 … 64 bits	the width is the contract
`layout packed . • magic u32 big .`	layout without padding · byte order	make bytes mean the same outside
`view wire_header b`	read bytes in that layout without copying	stops if length or alignment is off
`try_view wire_header b • encode wire_header h`	a view that gives `none` on failure · a value into bytes of that layout	at a boundary, the non-stopping one
`var a bitset 64 be bitset_new 64 . • add a 1 . • intersect a b`	a set of small numbers and its operations	a set, not the bits of a word

`view_segments back d · segs ss · seg ss i`	scattered pieces as one view without copying · piece count · piece i	removes the gathering copy
type words such as `byte` · `lock`	`W-NOT-YET` · `E-LOCK-NOTYET`	if there is no meaning, the tool says so

Table 13.2 — Named type syntax — shape · meaning · why it looks this way

Recap

`type` is an alias and `newtype` a new type with the same representation; `cast` crosses between them. `range lo hi` is a contract that has become the shape of a parameter: the caller keeps it and the body uses it as a fact. `cast` is the word for conversions that may change the value and stops when the value does not fit. `bits N` is an arbitrary-width integer. `layout packed`, byte-order marks and `view` match byte layout with the outside world.

Part IV — Contracts and effects

This part is where Lowent differs most from other languages. It unfolds the three things that actually hold up the goal set in chapter 1 — **knowing what an op can and cannot do from its head alone**.

Contracts (chapter 14) — write what an op receives and what it returns. Diagnostics decide whose fault it is, and enforced contracts become facts that remove run-time checks.

Effects (chapter 15) — write what an op does to the outside. Effects spread along calls, and purity becomes grounds for optimisation.

Capabilities (chapter 16) — write who permitted that work. Power is not scattered globally but handed over as arguments, and looking at the entry point alone tells you what outside the program can reach.

The last chapter (chapter 17) uses all three together, covering the design that places failure with `result` at the boundary and with contracts on the inside.

14 Contracts — write them, have them checked, lose the checks

In this chapter

14.1 A contract is a checked promise 173
14.2 Whose fault is it? 173
14.3 Contracts remove checks 175
14.4 Conditions over every element 175
14.5 Naming a contract 176
14.6 Decide now what can be decided now 177
14.7 Contract grades 178
14.8 Build modes decide the remaining checks 179
14.9 Common mistakes 180
14.10 This chapter's syntax at a glance 183

What to know first

chapter 2, A first program · a contract stops on entry
chapter 9, Sequences · writing the length condition as a contract removes bounds checks
chapter 11, Types that hold answers · an `errors` clause is a contract on the way out

Looking back

What did the single line `requires le n (len a) .` do in chapter 9's `sum_first`? And why must an index itself not use `le`?

> **A.** It gathered the check into one check on entry and let the bounds check on `index a i` inside the loop be removed. With `le` on an index, `i = len a` would be allowed, and that is one past the end, so indices use `lt`. This chapter covers all of it: what that contract is, who keeps it, and when it is checked and when it disappears.

The need for this chapter, and its context

Part IV is where this language earns its name. The goal of knowing what an op does from its head alone (chapter 1) is carried by three things: contracts, effects and capabilities. The first of them, contracts, has already appeared in pieces — stopping on entry, removing bounds checks, promising errors. This chapter gathers those pieces into one system. If effects (chapter 15) and capabilities (chapter 16) are about "what it does", contracts are about "what it takes and what it gives back".

By the end of this chapter

You will learn whose responsibility `requires`, `ensures` and `errors` each are, when they are checked, and how diagnostics assign blame when they break. You will pick up the principle by which contracts remove checks, conditions over every element of a slice (`elem_le` and so on), and naming contracts with `contract` and `satisfies`. You will also see violations decided at translation time, declarations of errors that can never happen, contract grades (`static`, `debug`, `assume`) and how build modes treat the checks that remain.

The questions this chapter answers

1. `ensures` is the op checking itself — how does that differ from a test?

14.1 A contract is a checked promise

A **contract** is the promise an op makes about its own inputs and outputs. Unlike a comment, it is checked.

examples/ch14/pair.low

```
module pair .
rem run: read_pair [1,2,3]

fn read_pair input data slice u8 . output u32 .
  requires ge (len data) 2 .
  ensures le ret 65535 .
do
  let hi u32 be widen u32 (index data 0) .
  let lo u32 be widen u32 (index data 1) .
  return add (mul hi 256) lo .
end
```

Output

```
$ lowentc --run read_pair pair.low [1,2,3]
read_pair([1,2,3]) = 258
  arg0 (written) = [1,2,3]
```

Read aloud, the head says: “this op must be given a byte sequence of length at least 2 (`requires`), promises that the value it returns is at most 65535 (`ensures`), and is pure (`fn`).” You know that much without opening the body. In `ensures`, `ret` stands for the returned value.

A contract is used in three places. The processor uses it **as a fact** to remove checks. A contract the processor could not prove is checked **at run time** and stops when broken. And it tells the reader what must be kept to call this op.

14.2 Whose fault is it?

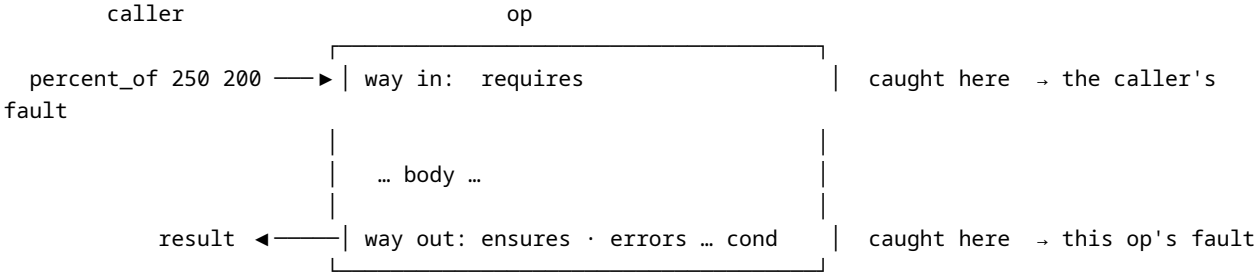
There are two places a contract breaks, and the diagnostic assigns blame.

Clause	When checked	Whose fault when broken

`requires`	on entry	**the caller** — did not keep the condition
`ensures`	on exit	**this op** — broke its own promise
`errors … <condition>`	on exit	**this op** — the condition held but it did not produce the error

Table 14.1 — When a contract breaks

Draw a contract as an op's **two doors** and the blame is decided by which door caught it.



examples/ch14/blame.low

```
module blame .
rem run: percent_of 50 200
rem trap: percent_of 250 200
rem run: clamp_to_100 400
rem trap: clamp_to_100 120

fn percent_of input part u64 . input whole u64 . output u64 .
  requires le part whole .
  requires gt whole 0 .
  requires le whole 1000000 .
  ensures le ret 100 .
do
  return div (mul part 100) whole .
end

rem clamps only values above 150 and forgets 101 … 150
fn clamp_to_100 input n u64 . output u64 .
  requires le n 1000 .
  ensures le ret 100 .
do
  if gt n 150 . do
    return 100 .
  end
  return n .
end
```

Output

```
$ lowentc --run percent_of blame.low 50 200
percent_of(50, 200) = 25
$ lowentc --run clamp_to_100 blame.low 400
clamp_to_100(400) = 100
$ lowentc --run percent_of blame.low 250 200
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: `requires` violated at entry — the caller broke the contract
$ lowentc --run clamp_to_100 blame.low 120
```

```
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: `ensures` violated at exit — THIS op broke its own promise (the calle
r was told a lie)
```

`percent_of 250 200` has a part larger than the whole, so it is the caller's fault, and the VM says "the caller broke the contract". `clamp_to_100 120` is the fault of an op that forgot to clamp values from 101 to 150, and the VM says "THIS op broke its own promise". When a program stops, the diagnostic answers at once "did I call it wrong, or is that op built wrong?".

> **Q. `ensures` is the op checking itself — how does that differ from a test?**
>
> **A.** A test checks that the answer is right for a few inputs. `ensures` checks that the promise holds **on every call**. Even if the promise breaks outside the inputs a test chose, `ensures` stops on that call. And callers may use `ensures` as a fact: a caller of `clamp_to_100` may trust that the result is at most 100 and have the check in `narrow u8` removed. It is also why the development repository's tooling can test ops from their contracts alone, without expected outputs (chapter 31).

14.3 Contracts remove checks

The processor uses contracts as facts to narrow the ranges values can take. When those ranges show safety, it removes overflow, division by zero, narrowing and slice bounds checks. So in this language **writing honestly makes code faster**.

```
fn bare input a u8 . output u8 .
do
  return add a 1 .          rem the overflow check remains
end

fn proven input a u8 . output u8 .
  requires le a 200 .
do
  return add a 1 .          rem no check --- it cannot exceed 201
end
```

The two ops have the same body. The only difference is one line of contract, and that line removes a run-time check. What the processor thinks is this short:

```
requires le a 200 .     →  a is 0 … 200
add a 1                 →  the result is 1 … 201
does it fit u8?         →  201 ≤ 255   ⇒ cannot overflow   ⇒ drop the overflow check
```

The check did not vanish but was **moved** to one check on entry. If the caller calls with a constant, or proves the range with its own contract, even the entry check disappears.

There is one important rule. **A contract that is not enforced is not used as a fact.** Trusting without checking and removing checks is not being faster but being wrong. This rule returns below with grades and build modes.

14.4 Conditions over every element

A contract's condition is a pure expression. Loops are statements, so "every element is at most 9" cannot be written as a loop. There are words that say it instead.

examples/ch14/elems.low

```
module elems .
rem run: digit_sum [1,2,3,9]
rem trap: digit_sum [1,20]

fn digit_sum input ds slice u8 . output u64 .
  requires elem_le ds 9 .
do
  var s u64 be 0 .
  for d ds do
    set s (add s (widen u64 d)) .
  end
  return s .
end
```

Output

```
$ lowentc --run digit_sum elems.low [1,2,3,9]
digit_sum([1,2,3,9]) = 15
  arg0 (written) = [1,2,3,9]
$ lowentc --run digit_sum elems.low [1,20]
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: `requires` violated at entry — the caller broke the contract
```

`elem_le ds 9` means every element is at most 9. There are also `elem_lt`, `elem_gt` and `elem_ge`. The condition is checked once on entry, and the arithmetic in the body uses the fact that the elements are at most 9.

14.5 Naming a contract

When several ops require the same condition, give the condition a name.

examples/ch14/named.low

```
module named .
rem run: half 10
rem run: third 10

contract positive do
  requires ge a 1 .
end

fn half satisfies positive . input a u8 . output u8 .
do
  return div a 2 .
end

fn third satisfies positive . input a u8 . output u8 .
do
  return div a 3 .
end
```

Output

```
$ lowentc --run half named.low 10
half(10) = 5
$ lowentc --run third named.low 10
third(10) = 3
```

`contract positive do … end` is a named contract, and an op adopts it with `satisfies positive .` at the very front of its head. Repeating the same condition by hand in several places means fixing one and forgetting another; a name leaves one place to fix. `satisfies` comes **first** in the head because it says what the op is first (chapter 3).

14.6 Decide now what can be decided now

When a call breaks the callee's `requires` and both sides are constants, translation rejects it without waiting to run.

examples/ch14/impossible.low

```
module impossible .
rem expect: E-CONTRACT-IMPOSSIBLE

fn small input a u8 . output u8 .
  requires lt a 10 .
do
  return a .
end

fn caller output u8 .
do
  return small 200 .
end
```

Output

```
$ lowentc --check impossible.low
impossible.low:12:0 E-CONTRACT-IMPOSSIBLE: this call breaks the callee's `requires`, and
 BOTH SIDES ARE CONSTANTS — so it can be decided here, now. It used to compile green and
 trap at run time (E-VM-CONTRACT): a bit budget that does not fit (`slot + shard + gener
ation > word`) shipped as a runnable program. A contract that can be decided at compile
time IS decided at compile time (RFC-0104 §8-3)
```

As the diagnostic's story says, this once translated green and only stopped at run time. A program carrying a bit budget that did not fit shipped in runnable form. A contract whose answer is available now is decided now.

The mistake in the other direction is rejected too. Declaring an error for a condition that `requires` already excludes means that error can never happen.

examples/ch14/dead.low

```
module dead .
rem expect: E-CONTRACT-DEAD

enum div_error do
  by_zero .
```

```
end

fn safe_div input a u32 . input b u32 . output result u32 div_error .
  requires ne b 0 .
  errors by_zero eq b 0 .
do
  guard ne b 0 . else return error by_zero .
  return ok (div a b) .
end
```

Output

```
$ lowentc --check dead.low
dead.low:10:0 E-CONTRACT-DEAD: this declared error can never occur: `requires` already e
xcludes the condition, and the condition's inputs cannot change during the op (remove th
e `requires`, or remove the error — not both)
```

With `requires ne b 0 .`, `errors by_zero eq b 0 .` can never be true. If the declaration stays, callers write code to handle `by_zero`, and that code never runs. Code that never runs is never tested, and untested code is wrong someday. The diagnostic says to remove one of the two — either make it the caller's responsibility (`requires`) or have the op handle it itself (`errors`).

> **A common misconception. The more contracts you write, the safer**
>
> Contracts that overlap or promise impossible cases draw a wrong picture. Contracts must be **facts**. A `requires` and an `errors` fighting over the same condition blur who is responsible, which is why it is rejected. Good contracts are short and put responsibility in one place.

14.7 Contract grades

A grade on a single contract clause decides when and how its condition is treated. The shape is `requires <grade> <condition> .`.

Grade	When looked at	Meaning
(none)	whenever possible	Prove it if possible; otherwise leave it as the build mode says
`static`	at translation	The processor must prove it; if it cannot, translation fails
`debug`	at run time	Checked only while the build mode keeps checks
`assume`	never	Only written down. Not used as a fact

Table 14.2 — Contract grades

examples/ch14/grades.low

```
module grades .
rem run: bump_checked 10
rem trap: bump_checked 250
rem run: bump_assumed 10
rem trap: bump_assumed 255

fn bump_checked input a u8 . output u8 .
  requires le a 200 .
do
```

```
  return add a 1 .
end

fn bump_assumed input a u8 . output u8 .
  requires assume le a 200 .
do
  return add a 1 .
end
```

Output

```
$ lowentc --run bump_checked grades.low 10
bump_checked(10) = 11
$ lowentc --run bump_assumed grades.low 10
bump_assumed(10) = 11
$ lowentc --run bump_checked grades.low 250
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: `requires` violated at entry — the caller broke the contract
$ lowentc --run bump_assumed grades.low 255
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
```

The same condition is written two ways. `bump_checked` rejects 250 on entry and then removes the overflow check in `add a 1`. `bump_assumed` does not check on entry. In exchange it does not use the condition as a fact either, so the overflow check **remains**. That is why giving it 255 stops with overflow (`E-VM-OVERFLOW`), not with a contract violation.

`assume` is a place to write down what the processor cannot yet prove but a person knows. Written down, readers know it, and when the processor grows it can be raised to `static`. Treating an unenforced condition as a fact would mean removing checks that should not have been removed, and believing that was right when the condition turns out false. So `assume` is not a fact.

14.8 Build modes decide the remaining checks

Proven contracts leave no check in any mode. What a build mode decides is the treatment of contracts that **could not be proven**. The mode is written in the source as `build <mode> .`.

Mode	Remaining contract checks
`debug`	Kept; stop and say what broke
`test`	Kept like `debug`; tests are also built and run
`release_safe`	Kept; stop without a message
`release_fast`	**Removed**. Execution may continue with a broken contract

Table 14.3 — The four build modes

examples/ch14/fast.low

```
module fast .
rem run: bump 10
rem run: bump 250
```

```
build release_fast .

fn bump input a u8 . output u8 .
  requires le a 200 .
do
  return add a 1 .
end
```

Output

```
$ lowentc --run bump fast.low 10
bump(10) = 11
$ lowentc --run bump fast.low 250
bump(250) = 251
```

Under `build release_fast .`, `bump 250` does not stop and gives 251. The same source stops under `debug`. `release_fast` opens one more trust boundary, where **a person promises** the contracts are true. It is one reason the claim that this language has no undefined behaviour is limited to “the safe subset”. Removing checks for speed is a choice you may make, but the fact of choosing must stay in the source. That is why the mode is a `build` statement and not a command-line flag.

> **In practice. Entry checks the processor cannot build**
>
> Not every expression in a contract can become an entry check. The processor in this edition builds entry checks from `requires` of shapes like `name comparison constant`, `len`, `elem_*` and field paths, and reports `W-CONTRACT-IGNORED` when it meets an expression it cannot build. While writing this book it turned out that an `ensures` containing an expression (`ensures le (mul ret 2) n .`) was not checked, without any warning. An unenforced contract is not used as a fact either, so no wrong optimisation results, but the fact that a promise goes unchecked should be reported. That is a place where the processor ought to speak, so it is a defect of this edition.

14.9 Common mistakes

> **Counter-example. Calling the returned value `result` in `ensures`**
>
> **examples/ch14/mistake_result.low**
>
> ```
> module mistake_result .
> rem expect: E-ENS-UNDEF
>
> fn clamp input a u8 . output u8 .
>   rem ✘ the returned value is named `ret`, not `result`
>   ensures le result 100 .
> do
>   guard le a 100 . else return 100 .
>   return a .
> end
> ```
>
> **Output**

```
$ lowentc --check mistake_result.low
mistake_result.low:6:0 E-ENS-UNDEF: ensures references an undefined name — an ensures th
at names nothing checks nothing, and the interval analysis DERIVES the result range from
 it (`ret` is the result)
```

The returned value is named `ret`. `result` is already a type name (`result u8 e`), so it is not reused as a value name. An `ensures` that names something undefined checks nothing, yet the analysis might derive a result range from it, so this is an error (`E-ENS-UNDEF`), not a warning.

Counter-example. Writing an index precondition with `le`

examples/ch14/mistake_leindex.low

```
module mistake_leindex .
rem run: at [1,2,3] 2
rem trap: at [1,2,3] 3

fn at input xs slice u8 . input i u64 . output u8 .
  rem ✘ `le` allows `i = len xs` --- that slot is one past the end
  requires le i (len xs) .
do
  return index xs i .
end
```

Output

```
$ lowentc --run at mistake_leindex.low [1,2,3] 2
at([1,2,3], 2) = 3
  arg0 (written) = [1,2,3]
$ lowentc --run at mistake_leindex.low [1,2,3] 3
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

The slots of a slice of length 3 are 0, 1 and 2. `requires le i (len xs)` also allows `i = 3`, so the contract passes and the following `index` stops with `E-VM-BOUNDS` trying to read one slot past the end. When a contract is wrong, the stop moves from the contract into the body, and the diagnostic says “out of bounds” instead of “the caller’s fault”. Use `lt` for indexes; `le` is right for counts (“read `n` items”).

Counter-example. Writing preconditions that cannot hold together

examples/ch14/mistake_contradict.low

```
module mistake_contradict .
rem expect: E-CONTRACT-UNSAT

rem ✘ no value satisfies both conditions --- every call stops
fn pick input a u8 . output u8 .
  requires le a 100 .
  requires ge a 200 .
do
  return a .
end
```

Output

```
$ lowentc --check mistake_contradict.low
mistake_contradict.low:7:0 E-CONTRACT-UNSAT: two preconditions on the same input cannot
both hold, so EVERY call stops at the door and the body never runs. A contract that no a
rgument satisfies is not a strong contract, it is a dead op — usually one line left behi
nd when the other was edited. Keep the one you meant
```

No `u8` is both at most 100 and at least 200, so this op stops at entry however it is called. Usually one `le` or `ge` was written backwards, or an old line was not removed during an edit. It sits beside an error declaration that can never happen (`E-CONTRACT-DEAD`) and is rejected with `E-CONTRACT-UNSAT`: a contract no argument satisfies is not a strong contract but a dead op.

A common misconception. `requires` is a tool for validating user input

examples/ch14/contract_input.low

```
module contract_input .
rem run: check_age 30
rem trap: age_strict 200
rem run: age_checked 200

enum age_error do
  too_old .
end

rem contract: accept only values the caller has already checked --- breaking it stops th
e program
fn age_strict input a u8 . output u8 .
  requires le a 150 .
do
  return a .
end

rem values from outside may be wrong --- report it as a value
fn age_checked input a u8 . output result u8 age_error .
  errors too_old gt a 150 .
do
  guard le a 150 . else return error too_old .
  return ok a .
end

fn check_age input a u8 . output u8 .
do
  match age_checked a do
    case ok v . return age_strict v .
    case error e . return 0 .
  end
end
```

Output

```
$ lowentc --run check_age contract_input.low 30
check_age(30) = 30
$ lowentc --run age_checked contract_input.low 200
```

```
age_checked(200) = err too_old
$ lowentc --run age_strict contract_input.low 200
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: `requires` violated at entry — the caller broke the contract
```

When a contract breaks, **the program stops**. A contract promises "the caller has already checked", so a broken one means there is a bug in the code. A user typing an age of 200 is not a bug; it happens all the time. Check values from outside as `age_checked` does and report the problem as a `result`; pass only checked values to an op with a contract such as `age_strict`. Because the call comes after the check, the analysis removes the contract check inside `check_age`.

14.10 This chapter's syntax at a glance

Shape	Meaning	Why
`requires le a 200 .`	condition on entry — the caller's responsibility	gathers checks into one at entry and removes checks in the body
`ensures le ret 100 .`	promise on exit — this op's responsibility	`ret` is the returned value — callers use it as a fact
`errors too_big gt a 200 .`	promise to return this error under this condition	a contract on the way out — errors are promised too
`requires elem_le ds 9 .`	a condition on every element	contracts are expressions, so no loops
`contract positive do … end`	give a contract a name	one place to change a shared condition
`fn half satisfies positive . …`	adopt a named contract (first in the head)	say what the op is first
`requires static … • debug • assume`	contract grades	decide per clause when and how it is checked — `assume` is not a fact
`build release_fast .`	remove unproven contract checks	the choice of speed stays in the source

Table 14.4 — Contract syntax — shape · meaning · why it looks this way

Recap

`requires` is the caller's responsibility, `ensures` and `errors` the op's, and the diagnostic says whose fault it was. Enforced contracts become facts and remove overflow, bounds and division-by-zero checks. `elem_*` expresses conditions over every element, and `contract` with `satisfies` names contracts. Violations between constants are rejected at translation, and so are declarations of impossible errors. `assume` is not a fact; build modes decide the treatment of unproven contracts, and `release_fast` removes those checks.

15 Effects — the marks an op leaves on the world

In this chapter

15.1 Effects are closed words 185
15.2 Effects spread to the caller 186
15.3 Declared but not done is reported 187
15.4 The effects line is a set 188
15.5 `via` — a type argument decides the effects 189
15.6 What purity allows 189
15.7 Common mistakes 190
15.8 This chapter's syntax at a glance 194

What to know first

chapter 5, Ops · a `fn` is pure and a `proc`'s `effects` is a narrowing clause
chapter 14, Contracts · an unenforced promise is not used as a fact

Looking back

In chapter 5, the pure `helper` did not print by itself and only called `say`, yet it was rejected. Why?

> **A.** Because effects spread to the caller. `say` performs `io`, so `helper`, which calls it, performs `io` too, and a pure `fn` cannot perform effects. This chapter covers what effects are made of, and what happens when the declaration and reality diverge in either direction.

The need for this chapter, and its context

If a contract is a promise about values (chapter 14), an effect is a promise about **behaviour**. Does it read files, obtain memory, possibly stop, wait for another flow? Without these answers in the head, a caller must follow the body to the end. Lowent has effects written as a set of closed words and the compiler checks both directions. Capabilities (chapter 16) are the partner that writes **who allowed** the effect, so effects come first.

By the end of this chapter

You will learn that effects are a closed set of atoms and what each atom means. You will pick up that effects spread along calls, that doing more than declared is rejected and declaring what is never done is warned about. You will see the rules that follow from the effects line being a set, the closure by which `concurrent` brings `wait` with it, and `via`, which lets a type argument decide the effects. Finally you will understand what purity allows the compiler to do.

The questions this chapter answers

1 To leave one log line, must a whole pure op become a `proc`?

15.1 Effects are closed words

An **effect** is an influence an op has on the outside. Authors cannot invent new effects; the list is fixed by the language.

Atom	Meaning
`none`	No effect at all. The bottom
`io`	Exchanges data with the outside (files, connections, standard I/O)
`alloc`	Obtains or returns memory from a **fixed window**
`heap`	Obtains memory from a **growing root**. Exists only on machines with an operating system
`state`	Changes state held by a module or actor, or the caller's storage
`panic`	May stop the program
`atomic`	Performs indivisible reads and writes
`concurrent`	Completion depends on the progress of another flow
`wait`	Waits; wakes up eventually without anyone's help
`lock`	Takes a lock
`device`	Touches a device directly
`unsafe`	Does work the language cannot check
`page_fault` · `blocking` · `cancel` · `detach`	a flow stopping or being cut off — no primitive produces them yet

Table 15.1 — Effect atoms

The nine `io`, `alloc`, `heap`, `state`, `panic`, `atomic`, `concurrent`, `wait` and `unsafe` have primitives that actually produce them, and the compiler enforces them — a body that does it must say so, and saying it without doing it is reported. The other six (`lock`, `device`, `page_fault`, `blocking`, `cancel`, `detach`) have no primitive yet. They are accepted as declarations and **spread** to callers, but whether the body really does it cannot be asked. A word not on the list is rejected.

examples/ch15/undef.low

```
module undef .
rem expect: E-EFFECT-UNDEF

proc checked input x u64 . output u64 . effects crash .
do
  return x .
end
```

Output

```
$ lowentc --check undef.low
undef.low:4:0 E-EFFECT-UNDEF: unknown effect (the vocabulary is closed: none/alloc/heap/
io/wait/concurrent/lock/atomic/unsafe/device/page_fault/blocking/cancel/detach/panic/sta
te) — a typo here silently declares the op PURE
```

Quietly accepting a typo would make that op read as having no effect — the case the diagnostic describes as “a typo here silently declares the op PURE”.

15.2 Effects spread to the caller

Calling an op adds its effects to the caller’s. Performing more effects than declared is rejected.

examples/ch15/spread.low

```
module spread .
rem expect: E-EFFECT

proc log_line input out cap io . input msg slice u8 . output u64 . effects io .
do
  return write_out out 1 msg .
end

proc compute input out cap io . input x u64 . output u64 . effects panic .
  requires le x 1000 .
do
  let n u64 be log_line out "computing\n" .
  if eq x 0 . do
    panic "zero" .
  end
  return mul x 2 .
end
```

Output

```
$ lowentc --check spread.low
spread.low:11:1 E-EFFECT: this op performs `io`, which its `effects` clause does not dec
lare — add it to `effects …`, or stop calling what needs it
```

`compute` declares only `effects panic .`, but calls `log_line` and so also performs `io`. The diagnostic says “add it to `effects`, or stop calling what needs it”. Conversely, the layers effects do not spread into are clear too.

examples/ch15/layered.low

```
module layered .
rem run: main

fn celsius_to_f input c i64 . output i64 .
  requires ge c -1000 .
  requires le c 1000 .
do
  return add (div (mul c 9) 5) 32 .
end

proc print_digit input out cap io . input d u8 . output u64 . effects io .
  requires le d 9 .
do
  let buf slice u8 be "0123456789" .
  return write_out out 1 (subslice buf (widen u64 d) (add (widen u64 d) 1)) .
end
```

```
proc main input out cap io . output u8 . effects io .
do
  let f i64 be celsius_to_f 100 .
  let n u64 be print_digit out (narrow u8 (mod f 10)) .
  let m u64 be write_out out 1 "\n" .
  return 0 .
end
```

Output

```
$ lowentc --run main layered.low
2
main() = 0
```

`celsius_to_f` is a pure `fn` and can be called anywhere. `print_digit` performs `io`, so it can be called only from `main`, which declares `io`. The computing layer and the layer that touches the outside separate in the head. Good design keeps the pure layer wide and the effectful layer thin.

> **Q. To leave one log line, must a whole pure op become a `proc`?**
>
> **A.** Yes. An op that logs leaves a trace outside, so it is not pure, and the head must show it. There is a common way out, though. Keep the computation pure and **return the result**, and log from the outer layer that receives it. This shape, pushing effects outward, is also easy to test — the pure layer can be checked right away with `--run` and `test` without capabilities.

15.3 Declared but not done is reported

Declaring an effect that is never performed is also a problem, because a declared effect is a cost the caller must bear.

examples/ch15/over.low

```
module over .
rem expect: W-EFFECT-OVER

proc double input x u64 . output u64 . effects panic .
  requires le x 1000 .
do
  return mul x 2 .
end
```

Output

```
$ lowentc --check over.low
over.low:6:1 W-EFFECT-OVER: this op DECLARES `panic` but never PERFORMS it. A declared e
ffect is a cost the caller must budget for (a pure caller cannot call an `io` op). Drop
it, or — if a future version will perform it — say so (RFC-0007)
```

`double` says it may stop, but it never stops. Then every op calling it must declare `panic`, and pure `fn`s cannot call it — it exaggerated what it does. The diagnostic says to remove it, or to say so if a future version will actually perform it.

> **A common misconception. Declaring effects generously saves work later**
>
> Generous effects spread to every caller. Declaring one `panic` in advance makes every calling op declare `panic`, and computations that should be pure lose their purity. An op that loses purity also loses optimisations such as reordering and memoisation. Declare effects only as far as they are actually performed.

15.4 The effects line is a set

The effects line is a **set**, not a list. Writing the same word twice is rejected.

examples/ch15/dup.low

```
module dup .
rem expect: E-EFFECT-DUP

proc checked input x u64 . output u64 . effects panic panic .
do
  if eq x 0 . do panic "zero" . end
  return x .
end
```

Output

```
$ lowentc --check dup.low
dup.low:4:0 E-EFFECT-DUP: an effect atom appears more than once in the `effects` clause
— the row is a SET, not a list. A repeat says nothing the first one didn't, and the read
er has to decide it's noise (RFC-0057 E3)
```

The second `panic` says nothing the first did not. `none` cannot appear with any other atom.

examples/ch15/nonemix.low

```
module nonemix .
rem expect: E-EFFECT-NONE-MIX

proc checked input x u64 . output u64 . effects none panic .
do
  if eq x 0 . do panic "zero" . end
  return x .
end
```

Output

```
$ lowentc --check nonemix.low
nonemix.low:4:0 E-EFFECT-NONE-MIX: `none` sits in the `effects` clause ALONGSIDE a real
effect — `none` means this op performs no effects, so it cannot co-occur with one. Say t
he effects, or say none; not both (RFC-0057 E2)
```

An op cannot both have effects and have none; write the effects or write that there are none. And a `fn` does not even carry `effects none` (chapter 5).

Some effects bring others with them. Today there is one such rule — writing `concurrent` counts as writing `wait` too, since waiting for another flow's progress may mean being suspended. Two

sets are therefore compared after applying this closure. Writing `concurrent` means `wait` need not be written separately, but writing only `wait` and doing something `concurrent` is rejected.

15.5 `via` — a type argument decides the effects

Generic containers are in an awkward spot, because their effects depend on which allocator they receive. Opened on a bump allocator over borrowed bytes, the effect is just `state`; opened on an allocator standing on the growing heap, `heap` appears. Always declaring the largest effect any allocator could produce is a lie, and declaring only the smallest is a **hidden allocation**.

```
export proc append
  input comptime t type .
  input comptime a type .
  input g mut vec t a .
  input x t .
  output bool .
  effects state via a .
  requires allocs.byte_allocator a .
```

This is the head of `append` in the standard library's `vecgen`. `effects state via a .` means "the allocation-family effects (`alloc`, `heap`, `atomic`) declared by the ops of type `a` are also this op's declaration". Each monomorphised instance has its effects decided by that type. Generic containers return in chapter 34.

15.6 What purity allows

A `fn` — an op whose effects are `none` — satisfies three properties.

- The same inputs always give the same result.
- Changing the order of calls does not change the program's result.
- If its result is unused, it need not be called.

These three allow optimisations. Two pure calls with the same arguments can be reduced to one (common-subexpression elimination), results can be remembered (memoisation), unused calls can be removed, and calls that do not depend on each other can be reordered or split up. That these optimisations do not change a program's meaning is proven in Coq (chapter 44). Call boundaries and the timing of traps are outside that proof.

> **In practice. Where effect declarations become premises of a proof**
>
> The soundness theorem of the effect system says "the declared effects cover the effects that actually happen". That theorem is what the theorems on effect-based optimisation stand on. So the compiler rejects code that does **more** than declared (the theorem's premise) and warns about code that does **less** (harmless to the theorem, but pushing false costs onto callers). That is why the two directions carry different weight.

15.7 Common mistakes

Counter-example. Using `panic` in a pure `fn`

examples/ch15/mistake_fnpanic.low

```
module mistake_fnpanic .
rem expect: E-EFFECT-CALC

rem ✘ a pure `fn` uses `panic` --- stopping the program is an effect too
fn nonzero input x u64 . output u64 .
do
  if eq x 0 . do panic "zero" . end
  return x .
end
```

Output

```
$ lowentc --check mistake_fnpanic.low
mistake_fnpanic.low:6:1 E-EFFECT-CALC: this fn is declared pure but performs `panic` — m
ake it a `proc` with `effects …`, or remove the effect
```

Stopping the program leaves a mark on the outside too. If a caller memoises this op or reorders it, the moment of the stop changes, so a `fn` cannot use `panic` and is rejected with `E-EFFECT-CALC`. Attaching `effects panic .` to a `fn` is rejected with the same code — the words `fn` and `proc` already say whether an op is pure. There are two fixes: make it a `proc` that declares stopping as an effect, or move the condition into a contract, which makes it the caller's responsibility.

examples/ch15/fnpanic_fixed.low

```
module fnpanic_fixed .
rem run: nonzero_checked 5
rem trap: nonzero_strict 0

rem fix 1 --- a `proc` that declares stopping as an effect
proc nonzero_checked input x u64 . output u64 . effects panic .
do
  if eq x 0 . do panic "zero" . end
  return x .
end

rem fix 2 --- move the condition into a contract and it stays a `fn`
fn nonzero_strict input x u64 . output u64 .
  requires ne x 0 .
do
  return x .
end
```

Output

```
$ lowentc --run nonzero_checked fnpanic_fixed.low 5
nonzero_checked(5) = 5
$ lowentc --run nonzero_strict fnpanic_fixed.low 0
```

```
== ir diagnostics (1) ==
0:0 E-VM-CONTRACT: `requires` violated at entry — the caller broke the contract
```

The second fix is usually better. `nonzero_strict` stays pure, and the diagnostic says the fault of passing 0 lies with the caller.

Counter-example. A `fn` writing into a slice received as `mut`

examples/ch15/mistake_fnmutwrite.low

```
module mistake_fnmutwrite .
rem expect: E-EFFECT-PURITY

rem ✘ a `fn` writes into the caller's slice received as `mut`
fn clear input xs mut slice u8 . output void .
do
  var i u64 be 0 .
  while lt i (len xs) do
    set (index xs i) 0 .
    set i (add i 1) .
  end
end
```

Output

```
$ lowentc --check mistake_fnmutwrite.low
mistake_fnmutwrite.low:5:0 E-EFFECT-PURITY: a `fn` WRITES through a `mut` parameter — th
at write is visible to the CALLER. A fn is an enforced purity contract (SPEC-003 §27): c
allers may memoise it, reorder it, or elide it. An op that changes caller-owned storage
can do none of those. Declare it a `proc`. (Local mutation stays pure: storage confined
to the op is not observable — a machine write is not an observable effect. RFC-0057)
```

The zeros written by `clear` stay in the caller's slice. To the caller the outside has changed, and such an op cannot be memoised, removed or reordered. Hence `E-EFFECT-PURITY`. Change the head to `proc clear … effects state .`. As the diagnostic adds, changing locals inside an op does not break purity (see the misconception below).

Counter-example. Putting commas between effects

examples/ch15/mistake_effcomma.low

```
module mistake_effcomma .
rem expect: E-VOCAB-REMOVED

rem ✘ a comma between effects --- just separate the words with spaces
proc greet input out cap io . input x u64 . output u64 . effects io, panic .
do
  if eq x 0 . do panic "zero" . end
  return write_out out 1 "hi\n" .
end
```

Output

```
$ lowentc --check mistake_effcomma.low
5:68 E-VOCAB-REMOVED: `,` (R3) was removed — RFC-0103, 2026-08-27. It opened the NEXT op
erand of the same form, but nothing used it that way: every `,` in the corpus was a LINE
 CONTINUATION, and newlines no longer close a form, so continuing a line needs nothing a
t all. A form is `head operand*` and ends at its closer `.` — just write the operands, o
n as many lines as you like.
```

The habit of separating a list with commas becomes `E-VOCAB-REMOVED` here. The comma was once in the grammar but was only ever used to continue a line; newlines no longer close a form, so it was removed. Separate the words with spaces, as in `effects io panic .`. The form ends at the stop.

A common misconception. Using `var` and `set` makes an op impure

examples/ch15/pure_local.low

```
module pure_local .
rem run: sum_to 10

rem still pure with `var` and `set` --- the changes are visible only inside this op
fn sum_to input n u64 . output u64 .
  requires le n 1000 .
do
  var s u64 be 0 .
  var i u64 be 0 .
  while le i n do
    set s (add s i) .
    set i (add i 1) .
  end
  return s .
end
```

Output

```
$ lowentc --run sum_to pure_local.low 10
sum_to(10) = 55
```

`sum_to` keeps changing two locals, yet it is a `fn`. The storage that changes lives only inside this op; the caller sees only the result 55. It always gives the same answer for the same input, and not calling it leaves nothing behind. Purity does not mean “changes nothing inside”; it means “leaves no mark visible from outside”.

A common misconception. A `fn` never stops

examples/ch15/fn_can_stop.low

```
module fn_can_stop .
rem run: first [7,8]
rem trap: first []

rem a pure `fn`, yet it stops on an empty slice --- a bounds-check stop is not an effect
 the op performs
fn first input xs slice u8 . output u8 .
```

```
do
  return index xs 0 .
end
```

Output

```
$ lowentc --run first fn_can_stop.low [7,8]
first([7,8]) = 7
  arg0 (written) = [7,8]
$ lowentc --run first fn_can_stop.low []
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

What a `fn` cannot use is the `panic` **effect**. A stop raised by the processor because a contract or a bound broke — such as reading the first slot of an empty slice — is treated as the result of a wrong call, not as something the op did (canon 6.5.9). So the pure `first` also stops when given `[]`. To avoid the stop, expose the responsibility to the caller with `requires ge (len xs) 1 .`, or return an `option`.

Counter-example. Calling an op with a rare effect from a pure `fn`

examples/ch15/mistake_blocking.low

```
module mistake_blocking .
rem expect: E-EFFECT-CALC

rem blocking: the head says it may hold the flow of execution
proc wait_for_device output u64 . effects blocking .
do
  return 0 .
end

rem ✘ a pure fn calls an op that may hold the flow; in this edition it slips through
fn ready output u64 . do
  return wait_for_device .
end
```

Output

```
$ lowentc --check mistake_blocking.low
mistake_blocking.low:11:23 E-EFFECT-CALC: this fn is declared pure but performs (이름 없
는 효과 비트) — make it a `proc` with `effects …`, or remove the effect
```

`blocking` means “may hold the flow of execution”, so a `fn` calling such an op is not pure, and it is refused with `E-EFFECT-CALC`. Until 2026-09-16 six atoms — `lock`, `device`, `page_fault`, `blocking`, `cancel` and `detach` — did **not** spread to the caller, so this passed; `device`, which touches hardware directly, could hide behind a pure function. The same shape with `wait` was refused even then.

examples/ch15/blocking_wait.low

```
module blocking_wait .
rem expect: E-EFFECT-CALC
```

```
rem wait does spread; the same shape is refused with E-EFFECT-CALC
proc wait_for_device output u64 . effects wait .
do
  return 0 .
end

fn ready output u64 . do
  return wait_for_device .
end
```

Output

```
$ lowentc --check blocking_wait.low
blocking_wait.low:6:1 W-EFFECT-OVER: this op DECLARES `wait` but never PERFORMS it. A de
clared effect is a cost the caller must budget for (a pure caller cannot call an `io` op
). Drop it, or — if a future version will perform it — say so (RFC-0007)
blocking_wait.low:10:23 E-EFFECT-CALC: this fn is declared pure but performs `wait` — ma
ke it a `proc` with `effects …`, or remove the effect
```

These six have no primitive that performs them yet, so they are not asked about by "declared but never performed" (`W-EFFECT-OVER`) — there is no body to ask. They only propagate.

15.8 This chapter's syntax at a glance

Shape	Meaning	Why
`fn …`	no effects — write no `effects` clause	purity is visible in one word
`proc … effects io panic .`	the set of effects this op may perform	reading the head tells you what it does
`proc …` **(no clause)**	not narrowed — may perform anything	narrowing is the author's choice
atoms `io·alloc·heap·state·panic·⋯`	a closed list fixed by the language	so a typo never silently means "pure"
`effects panic panic .·none panic`	rejected (`E-EFFECT-DUP` · `E-EFFECT-NONE-MIX`)	a set has no repeats or contradictions
doing more than declared · declaring and not doing	error (`E-EFFECT`) · warning (`W-EFFECT-OVER`)	premise of the proofs · a false cost for callers
`concurrent`	brings `wait` along	waiting on another flow can block
`effects state via a .`	inherit the allocation effects of type argument `a`	each allocator gets exact effects

Table 15.2 — Effect syntax — shape · meaning · why it looks this way

Recap

Effects are a closed set of atoms, and words outside the list are rejected. Effects spread along calls; doing more than declared is rejected and declaring what is not done is warned about.

The effects line is a set, so duplicates and mixing with `none` are rejected, and `concurrent` brings `wait`. `via` lets a type argument decide allocation-family effects. Purity allows optimisations such as reordering, memoisation and removal.

16 Capabilities — power that is handed over

In this chapter

16.1 Kinds of capability 196
16.2 Capabilities travel down the chain 196
16.3 Effects and capabilities are a pair 198
16.4 Kind, not presence 200
16.5 What the entry point receives 200
16.6 Common mistakes 202
16.7 This chapter's syntax at a glance 205

What to know first

chapter 2, A first program · printing needs `input out cap io .`
chapter 15, Effects · effects are a closed set of atoms that spread along calls

Looking back

Chapter 15 said the effects line writes **what** an op does. What, then, can `effects io` alone not tell you?

> **A.** **What** the input/output is with. Writing to standard output, opening a file and opening a connection are not told apart by the single word `io`. The capabilities of this chapter narrow that down — an op that received `input fs cap file_system .` can open files but not connections.

The need for this chapter, and its context

In most languages any function can write to standard output, open files and read the clock. That power is scattered globally, so finding out "does this library touch the network?" means reading all its source. Lowent has no such ambient authority. Power is **handed over as an argument, like a value**. Effects (chapter 15) and capabilities write the same thing from two sides and check each other, and with this chapter the three pillars of Part IV (contracts, effects, capabilities) stand.

By the end of this chapter

You will learn the kinds of capability and the work each opens. You will pick up that capabilities are passed down call chains by name, and that looking at the entry point alone tells you what outside a program can reach. You will see the table pairing effects with capabilities, and the diagnostics for declaring an effect without a capability, handing over the wrong kind, and using a capability without writing it. You will also understand which capabilities an entry point may receive, and that capabilities cost nothing at run time.

The questions this chapter answers

1 Passing capabilities as arguments — doesn't that add an argument per call at run time and slow things down?

16.1 Kinds of capability

A capability is received as `input <name> cap <kind>` .. The kind decides what can be done with it.

Capability	What it opens
`cap io`	Standard input and output
`cap file_system`	Files and directories
`cap net`	Making connections and exchanging data
`cap tty`	Terminal size and modes
`cap clock`	Asking the time
`cap random`	Operating-system entropy
`cap args` · `cap env`	Program arguments · environment variables
`cap allocator`	Obtaining memory from a fixed window (does not grow)
`cap heap`	Obtaining memory from a growing root (only on machines with an operating system)
`cap atomic`	Atomic operations
`cap mmio`	Device registers
`cap c`	Calling C functions
`cap machine`	Emitting machine instructions directly

Table 16.1 — Capability kinds the processor gives meaning to

The reason for kinds is **least privilege**. With only one capability, the permission received to read files could also open connections, and the op's head would no longer say "what this op can do".

16.2 Capabilities travel down the chain

Passing a capability on needs no special notation. Write the received name like any other argument.

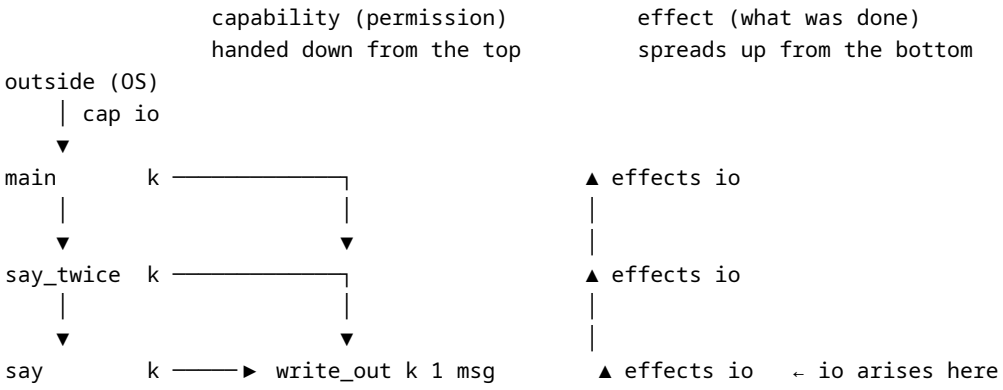
examples/ch16/chain.low

```
module chain .
rem run: main

proc say input k cap io . input msg slice u8 . output u64 . effects io .
do
  return write_out k 1 msg .
end

proc say_twice input k cap io . input msg slice u8 . output u64 . effects io .
do
  let a u64 be say k msg .
  let b u64 be say k msg .
  return add a b .
end

proc main input k cap io . output u8 . effects io .
do
  let n u64 be say_twice k "hi\n" .
  guard eq n 6 . else return 1 .
  return 0 .
end
```

Output

```
$ lowentc --run main chain.low
hi
hi
main() = 0
```

The chain looks like this — outside → `main`'s `k` → `say_twice`'s `k` → `say`'s `k` → `write_out`. Drawn together with effects, the two arrows run in opposite directions.



A capability must be handed down **from above** to be used; an effect arises **below** and is written in the head of every caller. So the head of any op shows both "what this op can do (effects)" and "who allowed it (capabilities)". An op that received no capability cannot call `say`, because it has nothing to pass. So "can this program touch files?" can be answered **by looking only at the entry point**. No op deep inside can secretly open a file: without the capability it cannot, and with it, the chain is visible all the way back to the entry.

> **Q. Passing capabilities as arguments — doesn't that add an argument per call at run time and slow things down?**
>
> **A.** It does not. A capability is **a mark at translation time**, not a run-time value. Ops that receive capabilities and ops that do not have the same signature at the C boundary. A capability asks "is this op allowed to be called?" at translation time, and nothing is left to ask at run time. The value that guards the boundary does not come back as a run-time cost.

16.3 Effects and capabilities are a pair

The effects line writes **what** an op does; capability inputs write **who allowed** it. The same thing is written from two sides, and each checks the other.

Effect	Authorising capability	When missing
io	`cap io` · `file_system` · `net` · `tty` · `clock` · `random`	`E-EFFECT-NO-CAP`
alloc	`cap allocator` (or a region)	`E-ALLOC-NOCAP`
heap	`cap heap`	`E-HEAP-NOCAP`
atomic	`cap atomic`	`E-ATOMIC-NOCAP`
device	`cap mmio`	`E-MMIO-NOCAP`
state · **panic** · **wait** · **concurrent**	written without a capability	—

Table 16.2 — Effects and the capabilities that authorise them

examples/ch16/pairs.low

```
module pairs .
rem run: main

proc main input out cap io . input al cap allocator . output u8 . effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 16 .
  guard is_some g . else return 1 .
  let n u64 be write_out out 1 "hi\n" .
  return 0 .
end
```

Output

```
$ lowentc --run main pairs.low
hi
main() = 0
```

`main` receives two capabilities. `cap io` authorises `effects io`, and `cap allocator` authorises `effects alloc`. `alloc_bytes al capacity 16` asks the allocator for 16 bytes and gives an `option`, since it may fail.

There are three ways the pair can go wrong. Declaring an effect with no authorising capability is rejected.

examples/ch16/nocap.low

```
module nocap .
rem expect: E-EFFECT-NO-CAP

proc main output u8 . effects io .
do
  let n u64 be write_out 1 "hello\n" .
  return narrow u8 n .
end
```

Output

```
$ lowentc --check nocap.low
nocap.low:4:0 E-EFFECT-NO-CAP: this op declares the `io` effect but receives NO capabili
ty that authorizes it. `io` is a cap-effect (RFC-0007 §6.7): I/O is a RIGHT you are HAND
ED, not an ambient power — `input fs cap file_system .` (or another `cap …` input). An e
ffect you declare but hold no capability for is a claim the signature cannot back
```

Allocation is the same: it declares using memory but not where the memory comes from.

examples/ch16/alloc_nocap.low

```
module alloc_nocap .
rem expect: E-ALLOC-NOCAP

proc scratch output u8 . effects alloc .
do
  return 1 .
end
```

Output

```
$ lowentc --check alloc_nocap.low
alloc_nocap.low:4:0 E-ALLOC-NOCAP: this op declares the `alloc` effect but receives NO a
llocation capability. There is no ambient heap in this language (RFC-0043 D1): to alloca
te you must be HANDED the right — `input a cap allocator .` or `input r region <name> .
.` (a region IS an arena allocator, RFC-0043 D7). An allocation nobody granted is exactl
y the hidden dependency this model removes
alloc_nocap.low:5:1 W-EFFECT-OVER: this op DECLARES `alloc` but never PERFORMS it. A dec
lared effect is a cost the caller must budget for (a pure caller cannot call an `io` op)
. Drop it, or — if a future version will perform it — say so (RFC-0007)
```

Receiving a capability and never handing it to the place that uses it is rejected too. A builtin op that requires a capability takes it as its **first operand**.

examples/ch16/missing.low

```
module missing .
rem expect: E-CAP-MISSING

proc hello input out cap io . output u64 . effects io .
do
  return write_out 1 "hello\n" .
end
```

Output

```
$ lowentc --check missing.low
missing.low:6:0 E-CAP-MISSING: `write_out` writes to the process output and needs the `c
ap io` value the entry received as its FIRST operand — output is a RIGHT you are handed
(RFC-0030 D2), never ambient. There is no hidden stdout
```

There is no way to reach a capability without naming it — in other words, there is no hidden standard output.

16.4 Kind, not presence

Holding some capability does not open every door.

examples/ch16/kind.low

```
module kind .
rem expect: E-CAP-KIND

proc stamp input out cap io . output u64 . effects none .
do
  return time_now out .
end
```

Output

```
$ lowentc --check kind.low
kind.low:6:0 E-CAP-KIND: this host leaf was handed a capability of the WRONG KIND. Autho
rity is by KIND, not by mere presence: an unrelated right (io, net, …) does not authoriz
e the clock, the filesystem, the socket or the OS entropy (RFC-0083 L3 · RFC-0077 §P1-2)
. Take the right this leaf names — the FFI boundary has always been checked this way (E-
FFI-CAPKIND); the host leaves used to be refused only at lowering, where a WRONG RIGHT w
as reported as an UNSUPPORTED FEATURE
```

`time_now` is the primitive that reads the clock. A `cap io` was handed over, so it is rejected as the wrong kind. Capabilities authorise by **what they are**.

> **A common misconception. Reading the time with `cap clock` requires declaring the `io` effect**
>
> The standard library's `clock.now_ms` takes a `cap clock` but is `effects none`. A clock breaks determinism, so it needs a capability, but it leaves no trace outside. Capabilities and effects are not always one-to-one. The `io` row of the pair table is a rule in one direction: "to perform the `io` effect you must receive one of these".

Authors can name capabilities of their own. Receiving your own capability as a parameter, as in `input logger cap audit .`, lets you build ops that only code holding that capability can call. What the processor gives meaning to are the kinds in the table above, and work of those kinds requires those kinds.

16.5 What the entry point receives

The op a program starts from receives **only capabilities**. It does not receive data, because nobody is there to hand it over.

examples/ch16/entry.low

```
module entry .
rem expect: E-ENTRY-PARAMS

proc main input out cap io . input n u64 . output u8 . effects io .
do
  return narrow u8 n .
end
```

Output

```
$ lowentc --check entry.low
entry.low:4:0 E-ENTRY-PARAMS: the entry point takes CAPABILITY inputs only (`input a cap
 args .`). A data parameter here would be a magic argv — RFC-0030 D2 rejects that: argum
ents are a RIGHT the runtime hands you, queried via `count`/`arg` on the cap
entry.low:5:1 W-EFFECT-OVER: this op DECLARES `io` but never PERFORMS it. A declared eff
ect is a cost the caller must budget for (a pure caller cannot call an `io` op). Drop it
, or — if a future version will perform it — say so (RFC-0007)
```

There is no magic argument like C's `argv`; program arguments are received through the `cap args` capability too. The capabilities an entry point may receive form a closed list — `args`, `env`, `io`, `allocator`, `heap`, `file_system`, `net`, `tty`, `clock`, `random`, `atomic`. Other capabilities (`mmio`, `machine`, `c`) are not something the executing environment can hand over, so an entry point requiring them is rejected with `E-ENTRY-CAP`. Such capabilities must be created where someone is entitled to grant them and flow in as arguments (chapters 29 and 30).

> **In practice. Finding out whether a dependency reaches the network**
>
> What worries you most when bringing in someone else's library is what that code touches. In Lowent you look through the heads of the library's `export`ed ops for any that take `cap net` (also look for `cap c`, which escapes into C, and `cap machine`, for machine code — the processor cannot see inside those). If there are none, the library cannot open connections. If there are, you must hand over `cap net` where you call them, so that place is the one to audit. A common supply-chain attack — imported code quietly connecting outward — becomes visible in the grammar.

Environment variables and operating-system randomness are also received as capabilities.

examples/ch16/envrandom.low

```
module envrandom .
rem run: main

use random .

rem environment variables need cap env and the operating system's entropy needs cap rand
om; both must be handed over
proc main input e cap env . input r cap random . input al cap allocator . output u8 . ef
fects alloc .
do
  let g option mut slice u8 . . be alloc_bytes al capacity 8 .
  guard is_some g . else return 250 .
  var buf mut slice u8 . be some_value g .
```

```
  rem fill eight bytes with operating-system randomness; the value differs every run, so
 it is not returned
  let n u64 be random.bytes r buf .
  rem a missing variable gives none, an absence rather than a value
  let v option slice u8 be env_get e "LOWENT_MANUAL_DEMO" .
  guard is_some v . else return narrow u8 n .
  return 99 .
end
```

Output

```
$ lowentc --run main envrandom.low
main() = 8
```

`env_get e "…"` needs `cap env`, and `random.bytes r buf` needs `cap random`. An environment variable is a channel that quietly changes behaviour from outside the program, and randomness is a channel that changes the answer for the same input. Both must be visible in the head so that the head alone tells whether "this program's answer is decided by its inputs". With no variable, `env_get` gives `none` and the example returns 8, the number of bytes filled. When tests need repeatable random numbers, use `rng_next` (the next number from a seed), which needs no capability.

16.6 Common mistakes

> **Counter-example. Thinking a `fn` may print because it received a capability**

examples/ch16/mistake_fncap.low

```
module mistake_fncap .
rem expect: E-EFFECT-CALC

rem ✘ receiving a capability does not let a `fn` produce output
fn say input out cap io . output u64 .
do
  return write_out out 1 "hi\n" .
end
```

Output

```
$ lowentc --check mistake_fncap.low
mistake_fncap.low:6:1 E-EFFECT-CALC: this fn is declared pure but performs `io` — make i
t a `proc` with `effects …`, or remove the effect
```

A capability says "who allowed it"; an effect says "what it does". Neither stands in for the other. Even with `cap io`, printing is the `io` effect, and a pure `fn` cannot perform effects, so this is `E-EFFECT-CALC`. Change it to `proc say … effects io .`. Both the permission (capability) and the action (effect) must be in the head, so that each checks the other.

> **Counter-example. Moving a received capability into a local name**

examples/ch16/mistake_caplocal.low

```
module mistake_caplocal .
rem expect: E-CAP-LOCAL

proc say input out cap io . output u64 . effects io .
do
  rem ✘ the capability is moved into a local name --- pass the received name directly
  let k cap io be out .
  return write_out k 1 "hi\n" .
end
```

Output

```
$ lowentc --check mistake_caplocal.low
mistake_caplocal.low:8:0 E-CAP-LOCAL: `k` is a LOCAL that a `cap io` was copied into. A
capability is not a value you carry in a local name: only a parameter of this op names a
 right it was handed, so hand the parameter itself to the leaf (RFC-0030 D2)
```

A capability is passed down the chain **under the name it was received with**. Moving it into a local name makes the chain look broken, so it is rejected with `E-CAP-LOCAL`; the message names the local and the kind that was copied into it. Write the parameter name directly, as in `write_out out 1 …`.

Counter-example. Writing to standard output with a capability of another kind

examples/ch16/mistake_fscapout.low

```
module mistake_fscapout .
rem expect: E-CAP-KIND

rem ✘ tries to write to standard output with the file capability
proc main input fs cap file_system . output u8 . effects io .
do
  let n u64 be write_out fs 1 "hi\n" .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_fscapout.low
mistake_fscapout.low:7:0 E-CAP-KIND: this leaf needs a `cap io`, and `fs` is a `cap file
_system`. One capability never stands in for another: each names a different right, and
holding one says nothing about the other (RFC-0011)
```

`cap file_system` opens files and directories; it is not the capability for standard output. Capabilities are authorised **by kind, not by possession**, so this is rejected with `E-CAP-KIND`, which prints the kind required beside the kind handed over. The fix: the entry receives `input out cap io .` and passes that name.

Counter-example. Passing a number where a capability belongs

examples/ch16/mistake_capforge.low

```
module mistake_capforge .
rem expect: E-CAP-FORGE

proc say input k cap io . output u64 . effects io .
do
  return write_out k 1 "hi\n" .
end

rem ✘ receives no capability at all, yet prints by passing the number 0 where a capabili
ty belongs
proc start output u8 .
do
  let n u64 be say 0 .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_capforge.low
mistake_capforge.low:12:0 E-CAP-FORGE: a LITERAL was passed where a capability is taken.
 A capability is handed over, never conjured: it can only be a name you were given (an `
input … cap …`) or an actor's capability field (RFC-0030 D2). Passing a number here woul
d let an op that received NO right reach the outside, and then the entry point no longer
 tells what the program can touch. The placeholder `0` means something only at the tool'
s door (`--run`)
```

`start` receives no capability at all, so it should not be able to print. Passing the number 0 in the `cap io` position of `say` looks like a way around that rule, and it is refused with `E-CAP-FORGE`. Only a **name you were handed** (`input … cap …`) or an actor's capability field may stand in a capability position. Without that, the promise of this chapter — "the entry point tells you everything the program can reach" — would fall to a single line. The placeholder `0` that `--run` uses means something only at the tool's door (chapter 31).

A common misconception. It is convenient to take capabilities and effects in advance, in case they are needed later

examples/ch16/capjustincase.low

```
module capjustincase .
rem expect: W-EFFECT-OVER

rem the capability and the effect are declared just in case they are needed later
proc area input out cap io . input w u64 . input h u64 . output u64 . effects io .
  requires le w 1000 .
  requires le h 1000 .
do
  return mul w h .
end
```

Output

```
$ lowentc --check capjustincase.low
capjustincase.low:8:1 W-EFFECT-OVER: this op DECLARES `io` but never PERFORMS it. A decl
```

```
ared effect is a cost the caller must budget for (a pure caller cannot call an `io` op).
 Drop it, or — if a future version will perform it — say so (RFC-0007)
```

`area` only multiplies, yet it declares `cap io` and `effects io`. Now every caller must find a `cap io` to pass, and pure code cannot call it at all. The tool warns with `W-EFFECT-OVER`. Take as few capabilities as possible — the list of received capabilities **is** "what this op can reach", so a generous list tells a lie.

16.7 This chapter's syntax at a glance

Shape	Meaning	Why
`input out cap io .`	receive the standard I/O capability	no ambient authority — power is handed in as an argument
`input fs cap file_system . · cap net · cap clock ···`	each kind opens different things	least authority — the head says what it can reach
`say k msg`	pass a received capability name as is	the chain is visible up to the entry point
`write_out out 1 "…"`	builtins that use a capability take it as the first operand	no way to reach a capability without a name
`effects io + cap io`	an effect and the capability that allows it	missing: `E-EFFECT-NO-CAP`
`effects alloc + cap allocator`	the pair for fixed-window allocation	missing: `E-ALLOC-NOCAP`
`proc main input … cap … . output u8 .`	the entry point receives only capabilities	nobody can hand it data
`input logger cap audit .`	a capability named by the author	only ops that received it can call the guarded op
`env_get e "…" · random.bytes r buf`	environment variable · OS randomness — `cap env` · `cap random`	outside channels that change the answer show in the head

Table 16.3 — Capability syntax — shape · meaning · why it looks this way

Recap

Capabilities are handed over as `input <name> cap <kind> .`, and the received name is passed on as is. Looking at the entry point shows what outside a program can reach. Effects and capabilities are a pair: declaring an effect requires receiving an authorising capability, and builtin ops that use a capability take it as their first operand. Capabilities authorise by kind and cost nothing at run time. The entry point receives only capabilities from a closed list.

17 Designing failure

In this chapter

17.1 The boundary and the inside 207
17.2 How to split an error enum 208
17.3 Moving up the layers 209
17.4 Where information may be thrown away 210
17.5 Where `panic` is acceptable 211
17.6 Common mistakes 211
17.7 This chapter's syntax at a glance 215

What to know first

chapter 11, Types that hold answers · three ways to report failure
chapter 14, Contracts · `requires` is the caller's responsibility, `errors` the op's
chapter 16, Capabilities · the entry point shows where the program touches the outside

Looking back

What question did chapter 11 use to separate `result`, `option` and `panic`?

> **A.** "What can the caller do?" If another path can be tried, `result`; if the thing is simply absent, `option`; if the caller already broke a promise and it cannot be fixed, stop. This chapter applies that question not to one op but to **every layer** of a program.

The need for this chapter, and its context

Knowing all the grammar does not by itself decide where and how failure should be reported. The same "invalid port number" must be a `result` in the layer reading a configuration file, and a contract in an inner layer that has already checked it. Get this wrong and checks repeat in layer after layer, or no layer checks at all. As the last chapter of Part IV, it lays out how to use contracts, errors and capabilities together to place failure.

By the end of this chapter

You will learn the design that separates values coming from outside from inner invariants, reporting with `result` at the boundary and with contracts inside. You will see the criteria for splitting error enums and what to put in `errors` clauses, and for deciding whether to pass failures up with `try` or handle them on the spot as they move up the layers. You will also see where it is fine to throw information away with `option`, and where `panic` is acceptable.

The questions this chapter answers

1. How does an error written without a condition, like `errors not_digit .`, differ from one with a condition, like `errors empty eq (len s) 0 .`?

17.1 The boundary and the inside

A program has **places where values come in from outside**: program arguments, file contents, bytes from the network, user input. Such values carry no promises. And there is an **inside** where values that passed a check flow around.

The first principle of the design is to separate the two.

Place	Nature of values	How to report
Boundary (parsing, input)	Anything can arrive	`result` and `errors`
Inside (computation)	Already passed a check	`requires` · `range` · `newtype`
No layer can handle it	A state that must not exist	`panic`

Table 17.1 — Where failure is reported

examples/ch17/parse.low

```
module parse .
rem run: parse_u16 [52,50]
rem run: parse_u16 []
rem run: parse_u16 [52,120]
rem run: parse_u16 [55,48,48,48,48]
rem run: port_or_default [56,48]
rem run: port_or_default [120]

enum parse_error do
  empty .
  not_digit .
  too_big .
end

fn is_digit input c u8 . output bool .
do
  return and (ge c 48) (le c 57) .
end

rem bytes from outside — anything can arrive, so failure is a value
fn parse_u16 input s slice u8 . output result u16 parse_error .
  errors empty eq (len s) 0 .
  errors not_digit .
  errors too_big .
do
  guard gt (len s) 0 . else return error empty .
  var acc u64 be 0 .
  for c s do
    guard is_digit c . else return error not_digit .
    set acc (add (mul acc 10) (widen u64 (sub c 48))) .
    guard le acc 65535 . else return error too_big .
  end
  return ok (narrow u16 acc) .
end

rem this layer needs no «why» — if absent, use the default
fn port_or_default input s slice u8 . output u16 .
do
  let r result u16 parse_error be parse_u16 s .
```

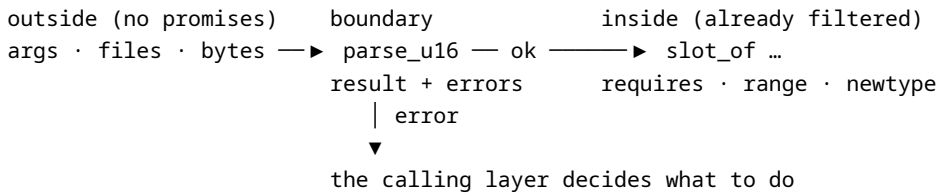
```
  guard not (is_error r) . else return 8080 .
  return ok_value r .
end
```

Output

```
$ lowentc --run parse_u16 parse.low [52,50]
parse_u16([52,50]) = ok 42
  arg0 (written) = [52,50]
$ lowentc --run parse_u16 parse.low []
parse_u16([]) = err empty
  arg0 (written) = []
$ lowentc --run parse_u16 parse.low [52,120]
parse_u16([52,120]) = err not_digit
  arg0 (written) = [52,120]
$ lowentc --run parse_u16 parse.low [55,48,48,48,48]
parse_u16([55,48,48,48,48]) = err too_big
  arg0 (written) = [55,48,48,48,48]
$ lowentc --run port_or_default parse.low [56,48]
port_or_default([56,48]) = 80
  arg0 (written) = [56,48]
$ lowentc --run port_or_default parse.low [120]
port_or_default([120]) = 8080
  arg0 (written) = [120]
```

`parse_u16` is a boundary. The bytes may be empty, may not be digits, may be too large. All three are **fixable failures**, so they are returned as a `result`. Demanding "digits only, please" with a contract would be the wrong design — the caller hands the bytes to this op as soon as it receives them, so it has no way to keep that demand.

Drawn as where a value comes from and where it goes:



> **Q. How does an error written without a condition, like `errors not_digit .`, differ from one with a condition, like `errors empty eq (len s) 0 .`?**
>
> **A.** With a condition, it becomes an outgoing contract: "under this condition, **exactly** this error happens". Returning normally while the condition holds is a contract violation. Callers can trust that avoiding the condition means they need not handle that error. Errors whose condition cannot be written as a contract expression (are all bytes digits?) are written without one and promise only "it can happen". Where a condition can be written, writing it tells callers more.

17.2 How to split an error enum

Give an error variant to **each case where the caller would act differently**. `empty`, `not_digit` and `too_big` were split because a different message can be shown for each. If callers do the same

thing for all of them, merge the variants; more variants mean more work for the code that handles them with `match`.

Conversely, avoid a giant enum that holds the errors of unrelated layers in one op. When file errors, parse errors and configuration errors are mixed into one enum, the `errors` clause in the head paints a wider picture than the op can actually produce, and declarations of impossible errors are rejected by translation (chapter 14).

17.3 Moving up the layers

A layer that receives a failure does one of two things: **pass it on** or **handle it there**.

examples/ch17/layers.low

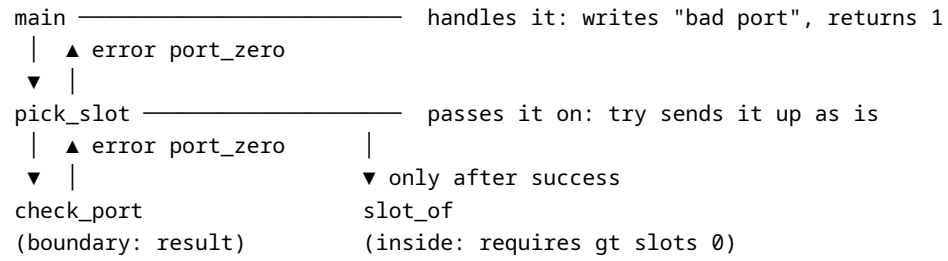
```
module layers .
rem run: main

enum config_error do
  bad_port .
  port_zero .
end

rem inner invariant — what the caller should already have filtered is a contract
fn slot_of input port u16 . input slots u64 . output u64 .
  requires gt slots 0 .
do
  return mod (widen u64 port) slots .
end

rem outside input — what can be wrong is a result
fn check_port input port u16 . output result u16 config_error .
  errors port_zero eq port 0 .
do
  guard ne port 0 . else return error port_zero .
  return ok port .
end

fn pick_slot input port u16 . output result u64 config_error .
  errors port_zero eq port 0 .
do
  let p u16 be try check_port port .
  return ok (slot_of p 16) .
end

proc main input out cap io . output u8 . effects io .
do
  let r result u64 config_error be pick_slot 8080 .
  guard not (is_error r) . else do
    let n u64 be write_out out 1 "bad port\n" .
    return 1 .
  end
  let m u64 be write_out out 1 "slot ok\n" .
  return narrow u8 (ok_value r) .
end
```

Output

```
$ lowentc --run main layers.low
slot ok
main() = 0
```

- `slot_of` is inside. If the number of slots were 0, the remainder operation would stop, but that is the caller's fault, so it is written as `requires gt slots 0 ..`
- `check_port` is close to a boundary. Port 0 means the configuration is wrong and can be fixed, so it is a `result`.
- `pick_slot` passes the failure up with `try`, and writes the same error in its own `errors`. After success it knows `p` is not 0 and calls the inner op.
- `main` is the outermost layer. There is nowhere further to pass the failure, so it **handles** it — writes a message and returns exit code 1.

The path a failure takes upward when `port` is 0. Going down, layers call; coming up, each passes the failure on or handles it.



As a rule: **if the nearest layer that could receive the failure can handle it, do not pass it on; if it cannot, pass it on.** `try` shrank the cost of passing it on to one word, but the fact that it is passed on stays in the caller's `errors` clause.

> **A common misconception. It is safest if every op returns a `result`**
>
> If even the inside returns `result`s, every call gets a `try` and every head an `errors`. Yet inside, the failure **cannot happen** — the boundary already filtered it out. Writing impossible failures creates handling code, and that code never runs. Write inner invariants with contracts and types (`range`, `newtype`). Then the check stays once at the boundary, and inside it is used as a fact and actually removed.

17.4 Where information may be thrown away

`port_or_default` does not need the **reason** for failure. Whatever went wrong, it uses the default port 8080. In such a layer, ask the `result` and ignore it, or turn it into an `option` with `try … else_none` and use `value_or`. Throwing information away is **that layer's decision**. If a lower layer throws it away early as an `option`, an upper layer that needs the reason has no way to get it back. So throw away as high up as possible.

17.5 Where `panic` is acceptable

Use `panic` when meeting a state that cannot be recovered from. The criterion is “which layer could handle this state?”.

- A program invariant is broken (an internal table is corrupted). No layer can produce a correct answer, so stop.
- Going on would cause greater harm (about to record data that failed verification).

Conversely, wrong user input, a missing file or a dropped connection are not reasons to `panic`; some layer can handle them. And `panic` is an effect, so the moment you use it, `effects panic` spreads to the head of that op and every op calling it (chapter 15). That cost is what makes you use `panic` sparingly.

> **In practice. The core of the HTTP request parser is rejection**
>
> The standard library’s `http` module reads HTTP/1.1 requests. The line its documentation introduces itself with is “its core is **rejection**”. Attacks such as request smuggling live where a server and a proxy **accept an ambiguous request differently**. So the parser accepts ambiguous input as little as possible and returns named errors. When a boundary op says clearly with a `result` what it rejects, the inside can trust the shape of the requests it accepted (chapter 36).

17.6 Common mistakes

> **Counter-example. Calling `panic` on bad input at the boundary**

examples/ch17/mistake_panicinput.low

```
module mistake_panicinput .
rem run: parse_or_die [52,50]
rem trap: parse_or_die [52,120]

rem ✘ one mistyped character from the user stops the whole program
proc parse_or_die input s slice u8 . output u64 . effects panic .
do
  var acc u64 be 0 .
  for c s do
    if or (lt c 48) (gt c 57) . do panic "not a digit" . end
    set acc (add (mul acc 10) (widen u64 (sub c 48))) .
    if gt acc 65535 . do panic "too big" . end
  end
  return acc .
end
```

Output

```
$ lowentc --run parse_or_die mistake_panicinput.low [52,50]
parse_or_die([52,50]) = 42
  arg0 (written) = [52,50]
$ lowentc --run parse_or_die mistake_panicinput.low [52,120]
== ir diagnostics (1) ==
```

```
0:0 E-VM-PANIC: the program called `panic` — this is an unrecoverable stop, and it is NO
T a contract violation (the code chose to stop, it did not break a promise)
```

The user only typed `4x`, and the whole program stops. Bad input is **something that happens all the time**, and some layer can deal with it — ask again, use a default, show a message. `panic` removes every one of those choices, and `effects panic` spreads to every caller on top of that. Return failure as a `result`, like `parse_u16` at the start of this chapter, and let the calling layer decide what to do.

Counter-example. Throwing the reason away in a lower layer

examples/ch17/mistake_dropwhy.low

```
module mistake_dropwhy .
rem run: explain []
rem run: explain [120]

enum parse_error do
  empty .
  not_digit .
end

fn parse_digit input s slice u8 . output result u8 parse_error .
  errors empty eq (len s) 0 .
  errors not_digit .
do
  guard gt (len s) 0 . else return error empty .
  let c u8 be index s 0 .
  guard and (ge c 48) (le c 57) . else return error not_digit .
  return ok (sub c 48) .
end

rem ✘ the lower layer threw the reason away early
fn digit_or_none input s slice u8 . output option u8 .
do
  return try (parse_digit s) else_none .
end

rem the upper layer wants different messages for empty and non-digit input, but sees 0 f
or both
fn explain input s slice u8 . output u8 .
do
  let d option u8 be digit_or_none s .
  guard is_some d . else return 0 .
  return 9 .
end
```

Output

```
$ lowentc --run explain mistake_dropwhy.low []
explain([]) = 0
  arg0 (written) = []
$ lowentc --run explain mistake_dropwhy.low [120]
```

```
explain([120]) = 0
  arg0 (written) = [120]
```

The moment `digit_or_none` turns the error into an `option` with `else_none`, the “why” is gone. The upper layer `explain` wants different messages for empty input and non-digit input, but sees 0 for both, and there is no way to get the reason back. Carry the reason all the way up and discard it only in the layer that knows it may be discarded.

examples/ch17/dropwhy_fixed.low

```
module dropwhy_fixed .
rem run: explain []
rem run: explain [120]
rem run: explain [55]

enum parse_error do
  empty .
  not_digit .
end

fn parse_digit input s slice u8 . output result u8 parse_error .
  errors empty eq (len s) 0 .
  errors not_digit .
do
  guard gt (len s) 0 . else return error empty .
  let c u8 be index s 0 .
  guard and (ge c 48) (le c 57) . else return error not_digit .
  return ok (sub c 48) .
end

rem carry the reason all the way up and split on it in the layer that needs it
fn explain input s slice u8 . output u8 .
do
  match parse_digit s do
    case ok v . return 9 .
    case error e . do
      rem bind the error value to a name, then split that enum once more
      match e do
        case empty . return 1 .
        case not_digit . return 2 .
      end
    end
  end
end
```

Output

```
$ lowentc --run explain dropwhy_fixed.low []
explain([]) = 1
  arg0 (written) = []
$ lowentc --run explain dropwhy_fixed.low [120]
explain([120]) = 2
  arg0 (written) = [120]
$ lowentc --run explain dropwhy_fixed.low [55]
```

```
explain([55]) = 9
  arg0 (written) = [55]
```

Split on an error variant directly with `case error <variant name>`.

examples/ch17/errvariant.low

```
module errvariant .
rem run: explain []
rem run: explain [120]
rem run: explain [53]

enum parse_error do
  empty .
  not_digit .
end

fn parse_digit input s slice u8 . output result u8 parse_error .
  errors empty eq (len s) 0 .
  errors not_digit .
do
  guard gt (len s) 0 . else return error empty .
  let c u8 be index s 0 .
  guard and (ge c 48) (le c 57) . else return error not_digit .
  return ok (sub c 48) .
end

fn explain input s slice u8 . output u8 .
do
  rem a declared variant name after `case error` matches that variant alone
  match parse_digit s do
    case ok v . return 9 .
    case error empty . return 1 .
    case error not_digit . return 2 .
  end
end
```

Output

```
$ lowentc --run explain errvariant.low []
explain([]) = 1
  arg0 (written) = []
$ lowentc --run explain errvariant.low [120]
explain([120]) = 2
  arg0 (written) = [120]
$ lowentc --run explain errvariant.low [53]
explain([53]) = 9
  arg0 (written) = [53]
```

When the name after `case error` is a **declared variant**, that arm takes that variant alone. When it is not, it binds the whole error value to that name (`case error e`). One spelling carries two meanings, but the rule is the one that holds everywhere else in a `match`: a bare name that is a variant is that variant (chapter 7). Write one arm per variant and the exhaustiveness check finds the one you forgot; bind a name instead and that single arm takes every error, so there is nothing left to find.

Counter-example. Calling an op that returns a `result` as a statement

examples/ch17/mistake_dropresult.low

```
module mistake_dropresult .
rem expect: W-RESULT-DISCARD

enum config_error do
  port_zero .
end

fn check_port input port u16 . output result u16 config_error .
  errors port_zero eq port 0 .
do
  guard ne port 0 . else return error port_zero .
  return ok port .
end

proc main input out cap io . output u8 . effects io .
do
  rem ✘ the result of the check is dropped as a statement --- the port is 0, yet "starte
d" is printed
  check_port 0 .
  let n u64 be write_out out 1 "started\n" .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_dropresult.low
mistake_dropresult.low:18:0 W-RESULT-DISCARD: this op returns a `result` — it says failu
re is a VALUE — and the value is dropped here, so a failure leaves no trace at all. Bind
 it and look at it (`let r … be …` then `is_error`), forward it (`try`), or say in the c
ode why the failure does not matter
```

`check_port 0 .` returned an error, but nobody received it: the port is 0, yet "started" would be printed and the exit code would be 0. The design of returning failure as a value only holds when the caller **looks at** that value, so the tool reports it as `W-RESULT-DISCARD`. Receive the `result` of such an op with `let` and ask, pass it on with `try`, or split it with `match`. Binding it and never reading it gets the same warning — the failure vanishes just the same. Where there genuinely is nothing to do with it (closing a handle on an error path), write `drop <name> .` to say you are letting this one go. The fault is not ignoring a failure; it is ignoring it **silently**.

17.7 This chapter's syntax at a glance

Shape	Meaning	Why
boundary op `output result t e . + errors`	return the failure of outside values as a value	the calling layer chooses what to do
inner op `requires · range · newtype`	invariants of already-filtered values	one check stays at the boundary; inside it is removed

`enum parse_error do empty . not_digit . end`	one variant per different caller action	more variants, more handling work
`let v u16 be try check_port port .`	pass it upward when you cannot handle it	the passing stays visible in `errors`
`case error e . do match e do … end end`	bind the error value, then split its variants	the name in `case error <name>` is a new binding unless it names a declared variant
`try … else_none • value_or`	discard the reason	discard as high up as possible
`proc … effects panic . + panic "…"`	stop in a state no layer can handle	not for bad input or missing files
entry point `output u8 .`	the outermost layer — handle failure and report by exit code	there is nowhere further to pass it

Table 17.2 — Shapes for designing failure — shape · meaning · why it looks this way

Recap

At boundaries where values come from outside, report failure with `result` and `errors`; inside, where checks have passed, write invariants with contracts and types. Give error variants to each case where the caller acts differently. A layer receiving a failure handles it if it can and passes it on with `try` if not. Throw information away as high up as possible, and use `panic` only for states no layer can handle.

Part V — Memory

18 Regions — where values live, and memory reclaimed all at once

In this chapter

18.1 Three places values live 219
18.2 Opening a region and obtaining space 220
18.3 Receiving a region 222
18.4 Nothing is carried out of a region 222
18.5 One cursor per root 223
18.6 The growing root 224
18.7 A stack on a region 225
18.8 Common mistakes 226
18.9 This chapter's syntax at a glance 231

What to know first

chapter 12, Borrowing · a borrow cannot outlive what it borrows
chapter 16, Capabilities · `effects alloc` pairs with `cap allocator`

Looking back

In chapter 12, what rejected returning a reference to a local from its op? And what were the ways to get a value out of an op?

> **A.** It was rejected with `E-ESCAPE`, because a local disappears when the op ends and the reference would point at nothing. To get a value out, return the value itself rather than a reference, or put it in storage the caller passed. This chapter covers where that "storage" comes from.

The need for this chapter, and its context

In a language without a garbage collector, someone must decide when memory is given back. C left it to people, who give it back twice or forget. Rust gives every value an owner and gives the memory back when the owner disappears. Lowent first puts a coarser unit in place — the **region**. Values born together and dying together go into one region, and the region is reclaimed all at once when it ends. Most memory in systems programs has this shape: temporary parser nodes, buffers used while handling one request. That is why Part V covers regions before individual ownership (chapter 19).

By the end of this chapter

You will learn the three places values live (local, static, obtained) and the two roots memory is obtained from (the fixed window and the heap). You will pick up how to open a region with a `region <name> <kind> do … end` block and obtain space with `alloc_bytes`, and how to pass a region as a parameter. You will see why carrying bytes from a region out of it, or carving from an outer region while an inner one is open, is rejected, and why the heap is refused on machines without an operating system.

The questions this chapter answers

1 Where do the values made by `make point do … end` or `some 7` live? Is that allocation?

18.1 Three places values live

A stored value lives in one of three places.

Kind	Description
Local	Born inside an op and gone when the op ends. The most common
Static	Present for as long as the program lives
Obtained	Obtained from a root. A region or ownership decides when it is given back

Table 18.1 — Where values live

What are usually called the stack and the heap correspond to local and obtained. The different names are because in this language they are **properties of values**, not the structure of the machine. And where a value lives is written in the source; the processor does not move things behind your back.

There are two roots memory is obtained from, carved and rewound separately.

Root	Carved with	Effect	Character
Fixed window	`cap allocator` · regions other than `heap`	`alloc`	**Does not grow**. `none` when exhausted
Heap	`cap heap` · `region <name> heap`	`heap`	**Grows**. Only on machines with an operating system

Table 18.2 — The two roots

The fixed window works even on machines without an operating system. There the window is the space between two bounds set by the linker (chapter 20).

Q. Where do the values made by `make point do … end` or `some 7` live? Is that allocation?

A. They live in the processor's **finite pool**. It is the op's implicit frame, so it has no effect and needs no capability. The pool is rewound on every loop iteration, and if more values are alive at once than the pool holds, execution stops on the spot. Its size is set by the machine and can be adjusted by whoever builds the program. This is why you do not write `alloc` every time you make a small aggregate value.

18.2 Opening a region and obtaining space

A region is opened with `region <name> <kind> do … end`. Inside the block, `alloc_bytes <region> capacity <n>` asks that region for `n` bytes.

examples/ch18/scratch.low

```
module scratch .
rem run: main

proc fill_count input n u64 . output u64 . effects alloc .
do
  region work arena do
    let g option mut slice u8 be alloc_bytes work capacity n .
    guard is_some g . else return 0 .
    let buf mut slice u8 be some_value g .
    var i u64 be 0 .
    while lt i (len buf) . do
      set (index buf i) 7 .
      set i (add i 1) .
    end
    var s u64 be 0 .
    for b buf do
      set s (add s (widen u64 b)) .
    end
    return s .
  end
  return 0 .
end

proc main input al cap allocator . output u8 . effects alloc .
do
  return narrow u8 (fill_count 10) .
end
```

Output

```
$ lowentc --run main scratch.low
main() = 70
```

- `region work arena do … end` opens a region. `arena` is the kind meaning “carve from the front and give back all at once”.
- `alloc_bytes work capacity n` gives an `option mut slice u8`. If there is not enough space, it is `none`. Running out of memory is a **value** too.
- `fill_count`’s head has `effects alloc`. Obtaining space is an effect; a pure `fn` cannot use a region.
- On **every path** out of the block — reaching the end, `return`, `break` or `continue` of an enclosing loop — the region is rewound. Even leaving with `return s .` reclaims the buffer. There is no code that gives it back piece by piece.

`main` receives `cap allocator`. The `alloc` effect of `fill_count` spreads up to it, so a capability authorising that effect is needed (chapter 16). A region block is entitlement to obtain space inside the op that opened it, but the effect still spreads to the caller.

A region’s kind is one of a closed eight — `stack`, `frame`, `arena`, `static`, `heap`, `mmap`, `disk`, `device`. Any other word is rejected.

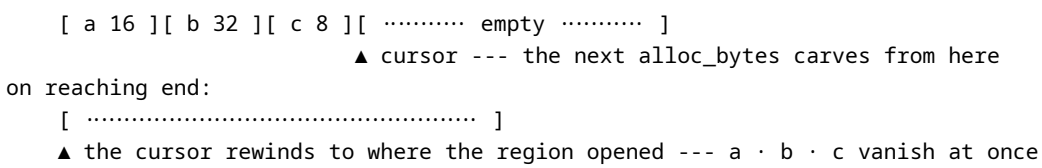
examples/ch18/kinds.low

```
module kinds .
rem run: carve_stack
rem run: carve_static

rem the kind word says what the region is for; in this edition only heap carves from the
 growing root, the rest from the fixed window
proc carve_stack output u64 . effects alloc .
do
  var n u64 be 0 .
  region r stack do
    let g option mut slice u8 . . be alloc_bytes r capacity 32 .
    if is_some g . do
      set n (len (some_value g)) .
    end
  end
  return n .
end

proc carve_static output u64 . effects alloc .
do
  var n u64 be 0 .
  region r static do
    let g option mut slice u8 . . be alloc_bytes r capacity 64 .
    if is_some g . do
      set n (len (some_value g)) .
    end
  end
  return n .
end
```

Output

```
$ lowentc --run carve_stack kinds.low
carve_stack() = 32
$ lowentc --run carve_static kinds.low
carve_static() = 64
```

A kind word leaves in the code **what the region is for**. In this edition only `heap` actually behaves differently — it carves from the growing root. The other seven all carve and rewind in the fixed window, so `carve_stack` and `carve_static` get 32 and 64 the same way. The kind is still written, and the list kept closed, for two reasons: the code need not change when the realisation is later tailored to a machine, and a word that could be anything would say nothing. Someone reading `region t arena` knows “carve from the front, give back all at once”.



This picture is why nothing leaks even though no value is given back one by one: giving back is moving one cursor.

18.3 Receiving a region

A region can also be passed as an argument. It is received as `input <name> region <type> .`, and the type is the name by which that region is called.

examples/ch18/param.low

```
module param .
rem run: main

type scratch u64 .

proc sum_squares input temp region scratch . input n u64 . output u64 . effects alloc .
  requires le n 100 .
do
  let g option mut slice u8 be alloc_bytes temp capacity n .
  guard is_some g . else return 0 .
  let buf mut slice u8 be some_value g .
  var i u64 be 0 .
  var s u64 be 0 .
  while lt i n . do
    set s (add s (mul i i)) .
    set i (add i 1) .
  end
  return s .
end

proc main input al cap allocator . output u8 . effects alloc .
do
  region r arena do
    let v u64 be sum_squares r 5 .
    return narrow u8 v .
  end
  return 1 .
end
```

Output

```
$ lowentc --run main param.low
main() = 30
```

`sum_squares` opens no region itself; it carves inside the region `r` the caller opened. So the lifetime of the buffer it received is decided by the caller's region. What is obtained from a region cannot outlive the region; when the region ends, what came from it ends too.

18.4 Nothing is carried out of a region

Putting bytes obtained from a region into a name outside the region is rejected.

examples/ch18/escape.low

```
module escape .
rem expect: E-REGION-ESCAPE

proc leak output u64 . effects alloc .
```

```
do
  var keep mut slice u8 be subslice "abcd" 0 0 .
  region r arena do
    let g option mut slice u8 be alloc_bytes r capacity 16 .
    if is_some g . do
      set keep (some_value g) .
    end
  end
  return len keep .
end
```

Output

```
$ lowentc --check escape.low
escape.low:10:0 E-REGION-ESCAPE: this value was allocated inside a `region` block and is
 being carried OUT of it. The block RECLAIMS its memory at `end` (SPEC-004 §4.5: the lif
etime IS the scope — that is why there is no `free`), so the value would point at bytes
the next allocation hands to somebody else. Copy what you need into memory that outlives
 the block, or move the block outward so it covers every use
```

When the region closes at `end`, those bytes are rewound, and `keep` outside would point at nothing. The places that carry things out are `return`, assignment to a name outside the region, and assignment to a field or element of such a name. Values that do not carry the region's bytes — integers and booleans such as `len buf` or a sum — may be carried out. That is what `fill_count` did when it returned a sum.

> **A common misconception. A region is just people managing lifetimes after all**
>
> People **decide** the lifetime, but translation **enforces** it. Paths by which a value from a region leaks out, and paths that point into a closed region, are blocked at translation. What people do is write "these values die together" as a block, and the block is visible. C's `free` calls are scattered and unchecked.

18.5 One cursor per root

Carving with the name of an outer region while a region of the same root is open inside is rejected.

examples/ch18/nested.low

```
module nested .
rem expect: E-ALLOC-NESTED

proc f output u64 . effects alloc .
do
  region outer arena do
    region inner arena do
      let g option mut slice u8 be alloc_bytes outer capacity 8 .
    end
  end
  return 0 .
end
```

Output

```
$ lowentc --check nested.low
nested.low:8:0 E-ALLOC-NESTED: this allocation names an OUTER source while a region of t
he SAME root is open inside it. Each root (the fixed window, the heap) has ONE cursor, s
o the inner region's `end` rewinds past these bytes and hands them to the next allocatio
n — the value would silently change under you (RFC-0112 D4). Allocate from the innermost
 region, move this allocation outside the inner block, or open the inner region on the o
ther root
```

The fixed window has a single place it carves from (a cursor). When the inner region ends and rewinds the cursor, bytes obtained in between with the outer name would be rewound with it. A different root — carving with a fixed-window capability inside a heap region — has its own cursor and is fine.

```
          ▼ where outer opened
                    ▼ where inner opened
          [ a ][ ⋯ ][ b ][ ✘ c ]
                    ▲ when inner ends the cursor rewinds to here → c vanishes too
  a = outer's · b = inner's · c = carved under the outer name while inner was open (✘ E-ALLOC-
NESTED)
```

For the same reason, an op spawned as a task cannot obtain memory directly from a root (`E-ALLOC-TASK`), because a cursor is not shared between flows. Sharing an allocator between flows needs an allocator that moves its cursor atomically (chapter 27).

18.6 The growing root

`region <name> heap` carves from the growing root.

examples/ch18/heap.low

```
module heap .
rem run: main

proc grow_twice output u64 . effects heap .
do
  region big heap do
    let a option mut slice u8 be alloc_bytes big capacity 4096 .
    let b option mut slice u8 be alloc_bytes big capacity 4096 .
    guard is_some a . else return 0 .
    guard is_some b . else return 0 .
    return add (len (some_value a)) (len (some_value b)) .
  end
  return 0 .
end

proc main input h cap heap . output u8 . effects heap .
do
  let n u64 be grow_twice .
  guard eq n 8192 . else return 1 .
  return 0 .
end
```

Output

```
$ lowentc --run main heap.low
main() = 0
```

The effect is `heap`, not `alloc`, and its partner capability is `cap heap`. A heap region is also reclaimed all at once when the block is left; it never moves bytes it has already handed out.

When building for a machine without an operating system, the heap cannot be requested through any door.

examples/ch18/heap_mcu.low

```
module heap_mcu .
rem flags: --target cortex_m
rem expect: E-HEAP-NOHOST

proc grow input h cap heap . output u64 . effects heap .
do
  let a option mut slice u8 be alloc_bytes h capacity 4096 .
  guard is_some a . else return 0 .
  return len (some_value a) .
end
```

Output

```
$ lowentc --check --target cortex_m heap_mcu.low
heap_mcu.low:5:0 E-HEAP-NOHOST: this op asks for the GROWING root — the `heap` effect, a
 `cap heap` input or a `region <name> heap` block — but the build target is FREESTANDING
 (`machine.no_heap`, e.g. cortex_m). A bare-metal board has no allocator that can hand o
ut more memory at run time. What it DOES have is the fixed window its linker script rese
rves: carve from that with `cap allocator` / `effects alloc` or any other region kind (R
FC-0112 D2 · RFC-0038 — the gate is on what the MACHINE cannot do, and only that)
```

`--target cortex_m` is a microcontroller without an operating system. The `heap` effect, `cap heap` inputs and `region … heap` blocks are all rejected. On the same machine, `alloc` carving from the fixed window is still available. Which root code stands on is written in its head, so whether a library runs on a machine without an operating system is answered by translation.

18.7 A stack on a region

A region hands out more than bytes. `stack_new <region> capacity <n>` makes a stack of `n` elements in that region. Use it for “take out what went in last first”, as when walking a tree or graph with a loop.

examples/ch18/stack.low

```
module stack_demo .
rem run: main

type scratch u64 .

rem take a stack from the region and pile up the digits --- the last one in comes out fi
rst
proc reverse_digits input temp region scratch . input n u64 . output u64 . effects alloc
 .
```

```
  requires le n 1000000 .
do
  let work stack u64 be stack_new temp capacity 8 .
  var v u64 be n .
  while gt v 0 . do
    push work (mod v 10) .
    set v (div v 10) .
  end
  var out u64 be 0 .
  var place u64 be 1 .
  while pop work into d . do
    set out (add out (mul d place)) .
    set place (mul place 10) .
  end
  return out .
end

proc main input al cap allocator . output u8 . effects alloc .
do
  region r arena do
    return narrow u8 (reverse_digits r 47) .
  end
  return 0 .
end
```

Output

```
$ lowentc --run main stack.low
main() = 74
```

- `let work stack u64 be stack_new temp capacity 8 .` makes the stack. It takes space from the region, so it is `effects alloc`.
- `push work x .` puts a value in.
- `while pop work into d . do … end` takes values out one at a time, binding each to `d`, as long as there is one. When the stack is empty the loop ends — the grammar has no way to stop by popping an empty stack.

Putting in the digits 7 and 4 of 47 brings them out as 4 and 7, giving 74. The stack is reclaimed together with the region, so there is no code to give it back.

18.8 Common mistakes

> **Counter-example. Taking the buffer out with `some_value` without asking whether space was granted**

examples/ch18/mistake_nocheck.low

```
module mistake_nocheck .
rem run: buf_len 16
rem trap: buf_len 1000000

proc buf_len input n u64 . output u64 . effects alloc .
do
```

```
  region work arena do
    rem ✘ takes it out without asking whether space was granted --- if the window is too
 small it stops taking out `none`
    let buf mut slice u8 be some_value (alloc_bytes work capacity n) .
    return len buf .
  end
  return 0 .
end
```

Output

```
$ lowentc --run buf_len mistake_nocheck.low 16
buf_len(16) = 16
$ lowentc --run buf_len mistake_nocheck.low 1000000
== ir diagnostics (1) ==
0:0 E-VM-NONE: some_value of none (panic)
```

16 bytes are granted, but a million bytes do not fit in the fixed window. `alloc_bytes` then gives `none`, and `some_value`, used without asking, stops with `E-VM-NONE`. It is the same mistake as not comparing C's `malloc` result with `NULL`, except that Lowent stops at the point of taking the value out instead of using space that does not exist. Ask first, as the examples in this chapter do with `guard is_some g . else return 0 .`. Running out of memory is a **value** to handle as well.

Counter-example. Returning a buffer obtained from a region

examples/ch18/mistake_returnbuf.low

```
module mistake_returnbuf .
rem expect: E-REGION-ESCAPE

proc make_buf input n u64 . output mut slice u8 . effects alloc .
do
  region work arena do
    let g option mut slice u8 be alloc_bytes work capacity n .
    guard is_some g . else return subslice "" 0 0 .
    rem ✘ returns bytes that are reclaimed when this block ends
    return some_value g .
  end
  return subslice "" 0 0 .
end
```

Output

```
$ lowentc --check mistake_returnbuf.low
mistake_returnbuf.low:10:0 E-REGION-ESCAPE: this value was allocated inside a `region` b
lock and is being carried OUT of it. The block RECLAIMS its memory at `end` (SPEC-004 §4
.5: the lifetime IS the scope — that is why there is no `free`), so the value would poin
t at bytes the next allocation hands to somebody else. Copy what you need into memory th
at outlives the block, or move the block outward so it covers every use
mistake_returnbuf.low:12:0 E-TYPE-RETMUT: this op declares a MUTABLE output (`mut slice`
/`mut_ref`) but RETURNS a read-only place — a shared parameter, a `let` without `mut`, a
 `ref X`, or a subslice/index/field of one. The caller would receive a mutable alias of
storage that is only held read-only, and could write through it — laundering the shared
/ fn-purity guarantee (measured: a pure fn mutated its shared input via such a return).
```

```
Return a mutable place — a fresh allocation, a `mut`/`owned` parameter, or a `var`/`mut`
 local — or drop `mut` from the output type
```

The bytes `make_buf` returns are rewound at `end`, so the moment the caller receives them the next allocation may hand that space to someone else. It has the same shape as returning the address of a local array in C, and it is rejected with `E-REGION-ESCAPE`. As the diagnostic says, move the block outward: the caller opens the region and passes it, and the receiving op carves from it.

examples/ch18/returnbuf_fixed.low

```
module returnbuf_fixed .
rem run: main

type scratch u64 .

rem the op that receives the region carves from it --- the caller's block decides how lo
ng the buffer lives
proc filled_sum input temp region scratch . input n u64 . output u64 . effects alloc .
  requires le n 100 .
do
  let g option mut slice u8 be alloc_bytes temp capacity n .
  guard is_some g . else return 0 .
  let buf mut slice u8 be some_value g .
  var s u64 be 0 .
  for b buf do
    set s (add s (widen u64 b)) .
  end
  return add s (len buf) .
end

proc main input al cap allocator . output u8 . effects alloc .
do
  region r arena do
    let v u64 be filled_sum r 5 .
    return narrow u8 v .
  end
  return 1 .
end
```

Output

```
$ lowentc --run main returnbuf_fixed.low
main() = 5
```

Counter-example. Writing an op that uses a region as a `fn`

examples/ch18/mistake_fnregion.low

```
module mistake_fnregion .
rem expect: E-EFFECT-CALC

rem ✘ written as a `fn` on the idea that nothing is left outside, since everything is re
turned at once
```

```
fn scratch_len input n u64 . output u64 .
do
  region work arena do
    let g option mut slice u8 be alloc_bytes work capacity n .
    guard is_some g . else return 0 .
    return len (some_value g) .
  end
  return 0 .
end
```

Output

```
$ lowentc --check mistake_fnregion.low
mistake_fnregion.low:6:1 E-EFFECT-CALC: this fn is declared pure but performs `alloc` —
make it a `proc` with `effects …`, or remove the effect
```

Everything is rewound when the block ends, so it looks as if nothing is left outside. Taking space is still the `alloc` effect: the result can depend on whether the window has room (`none`), and it overlaps with other code using the same window. Hence `E-EFFECT-CALC`. Write it as `proc … effects alloc ..`

Counter-example. Handing region bytes to an actor born outside the region

examples/ch18/mistake_outlives.low

```
module mistake_outlives .
rem expect: E-ALLOC-OUTLIVES

use allocs .

proc carve_inside output u64 . effects alloc state .
do
  var a allocs.bump_bytes be spawn actor allocs.bump_bytes .
  region r arena do
    let g option mut slice u8 . . be alloc_bytes r capacity 16 .
    if is_some g . do
      rem ✘ handing region bytes to an actor born outside the region
      let c u64 be send a init (some_value g) .
    end
  end
  return 0 .
end
```

Output

```
$ lowentc --check mistake_outlives.low
mistake_outlives.low:13:0 E-ALLOC-OUTLIVES: this hands bytes from a `region` block to an
 actor that was born OUTSIDE the block. The actor outlives the region, and nothing here
can see whether it keeps the slice — if it does, it will read bytes the region's `end` g
ave to somebody else (RFC-0112 D5). Create the actor inside the region, or give it memor
y that outlives it
```

The actor `a` was born before the region, so it lives on after the region closes. Whether it keeps the slice it receives in its state cannot be known at translation. If it does, it reads the old place

even after the region's `end` has given those bytes to someone else. So the check is conservative and refuses with `E-ALLOC-OUTLIVES`. Create the actor inside the region too, and both go away together.

examples/ch18/outlives_fixed.low

```
module outlives_fixed .
rem run: carve_inside

use allocs .

proc carve_inside output u64 . effects alloc state .
do
  var n u64 be 0 .
  region r arena do
    let g option mut slice u8 . . be alloc_bytes r capacity 16 .
    if is_some g . do
      rem the actor is born inside the region too, so both go away together
      var a allocs.bump_bytes be spawn actor allocs.bump_bytes .
      let c u64 be send a init (some_value g) .
      let p option mut slice u8 . . be send a reserve 10 .
      if is_some p . do
        set n (send a used) .
      end
    end
  end
  return n .
end
```

Output

```
$ lowentc --run carve_inside outlives_fixed.low
carve_inside() = 10
```

A common misconception. Space taken inside a loop is given back every round

examples/ch18/loop_region.low

```
module loop_region .
rem run: count_outer
rem run: count_inner

rem the region is outside the loop --- the 4096 bytes taken each round pile up until the
 block ends
proc count_outer output u64 . effects alloc .
do
  var got u64 be 0 .
  region work arena do
    var i u64 be 0 .
    while lt i 100 . do
      let g option mut slice u8 be alloc_bytes work capacity 4096 .
      if is_some g . do set got (add got 1) . end
      set i (add i 1) .
    end
  end
```

```
  return got .
end

rem the region is opened inside the loop --- it is rewound at the end of every round
proc count_inner output u64 . effects alloc .
do
  var got u64 be 0 .
  var i u64 be 0 .
  while lt i 100 . do
    region work arena do
      let g option mut slice u8 be alloc_bytes work capacity 4096 .
      if is_some g . do set got (add got 1) . end
    end
    set i (add i 1) .
  end
  return got .
end
```

Output

```
$ lowentc --run count_outer loop_region.low
count_outer() = 16
$ lowentc --run count_inner loop_region.low
count_inner() = 100
```

A region is rewound when its **block** ends, not when a round of the loop ends. In `count_outer` the region is outside the loop, so 4096 bytes pile up each round, and with this edition's default fixed window (65536 bytes) every request after the sixteenth gets `none`. `count_inner` opens the region inside the round and rewinds it every time, so all hundred requests succeed. For a buffer used only within one round, open the region inside the loop.

18.9 This chapter's syntax at a glance

Shape	Meaning	Why
`region work arena do … end`	open a region — rewound all at once on every way out of the block	no `free` — the lifetime is the block
`alloc_bytes work capacity n`	request `n` bytes from the region — `option mut slice u8`	running short is a value too
`effects alloc` · **`effects heap`**	take from the fixed window · take from the growing heap	the root in use is visible in the head
`input al cap allocator .` · **`cap heap`**	allocation capabilities received by the entry point	the pair that allows the effect
`input temp region scratch .`	receive a region the caller opened	the caller decides how long the buffer lives
`stack` **`frame`** **`arena`** **`static`** **`heap`** **`mmap`** **`disk`** **`device`**	the eight closed region kinds	words that mean something — others are `E-REGION-KIND`

storing into an outside name · returning	rejected (`E-REGION-ESCAPE`)	never point at reclaimed bytes
carving with an outer name while an inner region is open	rejected (`E-ALLOC-NESTED`)	one cursor per root
`--target cortex_m` + `heap`	rejected (`E-HEAP-NOHOST`)	a machine without an OS has no heap
`send` region bytes to an actor from outside	`E-ALLOC-OUTLIVES`	the actor may keep the bytes and outlive the region — create the actor inside

Table 18.3 — Region syntax — shape · meaning · why it looks this way

Recap

Values live in one of local, static and obtained, and the roots they are obtained from are the fixed window that does not grow (`alloc`) and the heap that does (`heap`). `region <name> <kind> do … end` opens a region, rewound all at once on every path out of the block. Regions can be passed as parameters. Bytes from a region cannot be carried out, and an outer region cannot be carved from while an inner region of the same root is open. On machines without an operating system the heap is rejected.

19 Ownership — one party responsible for disposal

In this chapter

19.1 `owned` and `drop` 233
19.2 Where branches meet 235
19.3 Release and completion 236
19.4 Not counting on the operating system to clean up 238
19.5 At what strength is memory safety guaranteed? 238
19.6 Common mistakes 239
19.7 This chapter's syntax at a glance 242

What to know first

chapter 12, Borrowing · many readers or one writer
chapter 18, Regions · a region is reclaimed all at once when it ends
chapter 17, Designing failure · someone has to receive a failure

Looking back

The regions of chapter 18 reclaim values all at once. Can resources that are disposed of **one at a time**, and whose closing **may fail** — file handles, say — be handled with regions alone?

> **A.** A region only rewinds bytes all at once; it cannot close files or flush buffers. And if closing fails, there is no place at a region's `end` to receive that failure. Such resources need **ownership**, where each value has a party responsible for disposing of it. That is this chapter.

The need for this chapter, and its context

The old names of memory defects are **double free** and **missed free**, and the newer one is **use after move**. All three arise when "who is responsible for disposing of this value now?" has more than one answer. Lowent checks at translation that there is always exactly one answer. It adds one more thing — if disposing **can fail**, it is not done silently. Regions (chapter 18) handled values that die together; this chapter handles values that die one by one.

By the end of this chapter

You will learn what `owned t` is and what `drop` does. You will pick up why reusing a moved value or disposing twice is rejected, and why differing ownership states across branches are rejected. You will see that disposal comes in two kinds, **release** (cannot fail) and **completion** (can fail), and why silently discarding a value that needs completion is rejected. You will also sort out at what strength this language's memory safety is guaranteed.

The questions this chapter answers

1 What happens when a value without ownership is passed?

19.1 `owned` and `drop`

`owned t` is a value that carries ownership. A value with ownership must be disposed of **exactly once**.

examples/ch19/sink.low

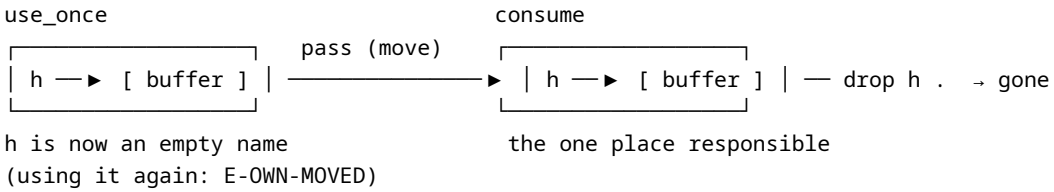
```
module sink .
rem run: use_once 5

type buffer u8 .

fn consume input h owned buffer . output u8 .
do
  drop h .
  return 0 .
end

fn use_once input v u8 . output u8 .
do
  var h owned buffer be v .
```

```
  return consume h .
end
```

Output

```
$ lowentc --run use_once sink.low 5
use_once(5) = 0
```

`consume` takes an `owned buffer` and disposes of it with `drop h .`. `use_once` makes an owned value with `var h owned buffer be v .` and passes it to `consume`. At that moment ownership **moves**. Now `consume` is responsible for disposing of `h`, and `use_once` can no longer use `h`.



Think of a locker with a single key. Once you hand the key over, you no longer hold it, and only the one person holding the key can empty (dispose of) the locker. So “two people empty it” (double free), “nobody empties it” (leak) and “open it again with the key you gave away” (use after move) cannot happen.

Using a moved value again is rejected.

examples/ch19/moved.low

```
module moved .
rem expect: E-OWN-MOVED

type buffer u8 .

fn consume input h owned buffer . output u8 .
do
  drop h .
  return 0 .
end

fn reuse input h owned buffer . output u8 .
do
  let a u8 be consume h .
  let b u8 be consume h .
  return add a b .
end
```

Output

```
$ lowentc --check moved.low
moved.low:15:0 E-OWN-MOVED: this `owned` value was already MOVED (consumed) — using it a
gain is use-after-move, which SPEC-004 §4.8 has always called a compile error and which
nothing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set <name> <new val
ue>` re-initialises the place — that is how a handle threads through a loop), or borrow
it LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered yet (E-IR-UNSU
P says so at the call site), so `f (ref h)` is not a way out today
```

Disposing twice is rejected too.

examples/ch19/twice.low

```
module twice_drop .
rem expect: E-OWN-MOVED

type buffer u8 .

fn twice input h owned buffer . output u8 .
do
  drop h .
  drop h .
  return 0 .
end
```

Output

```
$ lowentc --check twice.low
twice.low:9:0 E-OWN-MOVED: this `owned` value was already MOVED (consumed) — using it ag
ain is use-after-move, which SPEC-004 §4.8 has always called a compile error and which n
othing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set <name> <new valu
e>` re-initialises the place — that is how a handle threads through a loop), or borrow i
t LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered yet (E-IR-UNSUP
 says so at the call site), so `f (ref h)` is not a way out today
```

The diagnostic notes that the specification always called this an error and that for a while nothing enforced it. Today translation stops it.

> **Q. What happens when a value without ownership is passed?**
>
> **A.** It is copied. Values like `u64` or `point` are copied when passed, and the original name stays usable. Only owned values move. Large values whose copying cost worries you are passed by borrowing with `ref` (chapter 12). And when the place where a value-producing expression will land is empty, the expression cannot fail on the way, and the place belongs to that value alone, the value is built directly in place — no intermediate temporary and no moving copy. These three conditions can be counted from the source alone.

19.2 Where branches meet

Where an `if` splits and rejoins, the ownership state must be the same on every path.

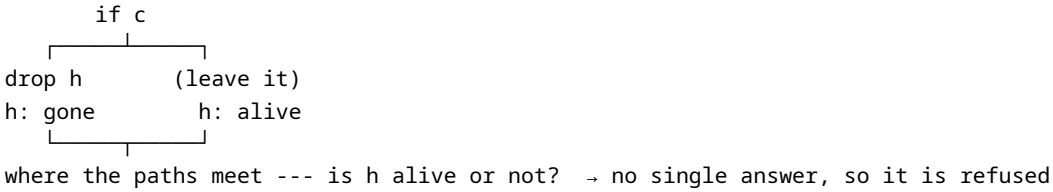
examples/ch19/join.low

```
module join .
rem expect: E-OWN-JOIN

type buffer u8 .

fn maybe input h owned buffer . input c bool . output u8 .
do
  if c . do
    drop h .
  end
  return 0 .
end
```

Output

```
$ lowentc --check join.low
join.low:8:0 E-OWN-JOIN: this `owned` value is CONSUMED on one path of this branch but s
till LIVE on another where they merge — Lowent requires the ownership state to be STATIC
ALLY consistent at a join (RFC-0044 §9.3): no hidden drop-flag decides it at runtime. Co
nsume it on EVERY path, or make it conditionally owned (`?owned`)
```

On the path where `c` is true, `h` was disposed of; on the false path it is still alive. After the paths meet, nobody can say whether `h` is alive. Some languages keep a hidden “already dropped” flag here, but Lowent requires the ownership state to be **statically** one thing. Dispose of it on both paths, or pass it on both paths.



By the same principle, moving one `owned` field of an aggregate and then moving the whole aggregate again is rejected (`E-OWN-PARTIAL`). The receiver thinks it got a whole aggregate, but one field inside already belongs to someone else.

19.3 Release and completion

Disposal comes in two kinds.

Kind	Examples	How it is handled
Release	Giving memory back	Cannot fail, so it may happen silently where the lifetime ends
Completion	Closing a file, flushing a buffer, committing a transaction	Can fail, so the author must write the call

Table 19.1 — The two kinds of disposal

Release has no failure to swallow. Completion, done silently, has nowhere to hand its failure. The criterion is one: **can finishing fail?**

kind of value	when passed	when it ends
no ownership (`u64` · `point` …)	copied; the original name stays usable	nothing to do
`owned`, needs release only	moved; the original name is emptied	released silently where its lifetime ends
`owned`, needs completion	moved; the original name is emptied	the author must write the completing call (otherwise `E-OWN-INCOMPLETE`)

Table 19.2 — when a value is passed and when it ends — at a glance

Which types require completion is declared by the program itself. If there is an op that takes the type as `owned` and returns a `result`, that is the declaration “finishing this can fail”. No new word is involved.

examples/ch19/complete.low

```
module complete .
rem run: session 3

type journal u64 .

enum flush_error do
  disk_full .
end

fn finish input j owned journal . output result u8 flush_error .
  errors disk_full eq j 0 .
do
  rem an empty journal means there was no room to write --- so this arm really does happ
en
  guard ne j 0 . else do
    drop j .
    return error disk_full .
  end
  drop j .
  return ok 0 .
end

fn session input n u64 . output result u8 flush_error .
  errors disk_full .
do
  var j owned journal be n .
  return finish j .
end
```

Output

```
$ lowentc --run session complete.low 3
session(3) = ok 0
```

Because `finish` takes an `owned journal` and returns a `result`, `journal` becomes a type needing completion. `session` finishes it by calling `finish j` and returns that (possibly failing) result as its own. Letting it go out of scope without calling the completion is rejected.

examples/ch19/incomplete.low

```
module incomplete .
rem expect: E-OWN-INCOMPLETE

type journal u64 .

enum flush_error do
  disk_full .
end

fn finish input j owned journal . output result u8 flush_error .
  errors disk_full .
```

```
do
  drop j .
  return ok 0 .
end

fn forget input j owned journal . output u8 .
do
  return 0 .
end
```

Output

```
$ lowentc --check incomplete.low
incomplete.low:11:0 W-ERRORS-UNRAISED: this `errors` clause names a failure the body nev
er returns. The clause is read as a promise about what this op can do, and the callers w
rite their handling from it — a branch that can never be taken is dead code the reader c
annot tell from live code. Return it (`return error <variant>`), forward one (`try`), or
 remove the clause
17:0 E-OWN-INCOMPLETE: this value is dropped automatically at the end of scope — but the
 program itself declares an op that takes this type `owned` BY VALUE and returns a `resu
lt`: finishing it CAN FAIL. An automatic drop is a RELEASE (total, non-suspending), and
it has nowhere to hand you that failure — it would SWALLOW it. A fallible finish (flush/
commit/close) is a COMPLETION and must be EXPLICIT: call it and handle the `result`. If
you really mean to discard the value and its failure, say so with `drop`. RFC-0058
```

As the diagnostic says, automatic disposal is a release, and a release has nowhere to return a failure, so it would **swallow** the failure. Finishing that can fail — flush, commit, close — must be written and called by the author.

> **A common misconception. It is convenient for a destructor to close the file automatically**
>
> A C++ destructor or Rust `Drop` closes a resource when it goes out of scope. What if closing fails? A destructor cannot return a value, so it ignores the failure or stops the program. That is where the fact that a full disk kept the last buffer from being written silently disappears. Lowent allows this convenience only for release and makes completion explicit. If you really mean to discard both the value and its failure, you **say you are discarding it** with `drop` — the diagnostic points that way too. Something vanishing silently and an author writing that it is discarded are different things.

19.4 Not counting on the operating system to clean up

Many programs lean on “the operating system cleans up when the process ends anyway”. Lowent does not. Because a forgotten disposal is caught at translation, the same code holds in environments without an operating system — firmware whose process never ends.

19.5 At what strength is memory safety guaranteed?

The words “memory safe” mean something only when they say what is safe and how. This language separates the strengths.

Rule	Strength	Meaning
Borrow exclusivity · no reference escape	static + proven	Stopped by translation; the rules' soundness is proven in Coq (sequential model)
Dispose exactly once · no region escape	static	Stopped by translation
Slice bounds · contracts	dynamic	Checked at run time; the check disappears when proven
Dangling generational handles	dynamic	Generations compared at run time (chapter 34)

Table 19.3 — Memory rules and their strength

So calling this language "fully statically safe" would be wrong. It is a design that mixes what is stopped statically, what is stopped at run time, and what is proven. What is proven, and what gap lies between the proofs and the compiler, is covered in chapters 42 and 50.

19.6 Common mistakes

Counter-example. Passing an owned value inside a loop

examples/ch19/mistake_loopmove.low

```
module mistake_loopmove .
rem expect: E-OWN-MOVED

type buffer u8 .

fn consume input h owned buffer . output u8 .
do
  drop h .
  return 0 .
end

fn three_times input v u8 . output u8 .
do
  var h owned buffer be v .
  var i u64 be 0 .
  while lt i 3 . do
    rem ✘ the second round passes `h` again after the first round moved it away
    let a u8 be consume h .
    set i (add i 1) .
  end
  return 0 .
end
```

Output

```
$ lowentc --check mistake_loopmove.low
mistake_loopmove.low:16:0 E-OWN-MOVED: this `owned` value is CONSUMED inside a LOOP and
never put back — the SECOND iteration would be a use-after-move. A loop body must end th
```

```
e way it began: consume it and REASSIGN the place (`set <name> …`, which re-initialises
it), or move the consumption out of the loop
```

Once `consume h` takes ownership in the first round, `h` in the second round already belongs to someone else. Translation does not need to follow the rounds one by one; it sees that "the loop body ends in a different shape than it began" and rejects it with `E-OWN-MOVED`. There are two fixes: move the consuming call out of the loop, or refill the moved place with a new value using `set`, so each round ends in the same shape.

examples/ch19/loopmove_fixed.low

```
module loopmove_fixed .
rem run: three_times 5

type buffer u8 .

fn consume input h owned buffer . output u8 .
do
  drop h .
  return 1 .
end

fn three_times input v u8 . output u8 .
do
  var h owned buffer be v .
  var n u8 be 0 .
  var i u64 be 0 .
  while lt i 3 . do
    set n (add n (consume h)) .
    rem refill the moved place with a new value --- the round ends in the same shape it
began
    set h v .
    set i (add i 1) .
  end
  drop h .
  return n .
end
```

Output

```
$ lowentc --run three_times loopmove_fixed.low 5
three_times(5) = 3
```

Counter-example. Moving a value while thinking you only read it

examples/ch19/mistake_readmove.low

```
module mistake_readmove .
rem expect: E-OWN-MOVED

type buffer u8 .

fn peek input h owned buffer . output u8 .
do
```

```
  rem ✘ taking the value into a name is a move too --- the `drop h` after it destroys a
moved value
  let v u8 be h .
  drop h .
  return v .
end
```

Output

```
$ lowentc --check mistake_readmove.low
mistake_readmove.low:10:0 E-OWN-MOVED: this `owned` value was already MOVED (consumed) —
 using it again is use-after-move, which SPEC-004 §4.8 has always called a compile error
 and which nothing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set <nam
e> <new value>` re-initialises the place — that is how a handle threads through a loop),
 or borrow it LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered yet
 (E-IR-UNSUP says so at the call site), so `f (ref h)` is not a way out today
```

`let v u8 be h .` does not look at `h`; it **moves** it into `v`. A value with ownership moves the moment it is stored under a name, so the later `drop h` tries to destroy a moved value. To only look, borrow it with `ref h` inside the same op. As the diagnostic notes, borrows across an op boundary are not lowered yet in this edition.

Counter-example. Believing a second name makes a copy

examples/ch19/mistake_twonames.low

```
module mistake_twonames .
rem expect: E-OWN-MOVED

type buffer u8 .

fn consume input h owned buffer . output u8 .
do
  drop h .
  return 0 .
end

fn both input h owned buffer . output u8 .
do
  rem ✘ believed a second name makes a copy --- ownership moved to `h2`
  var h2 owned buffer be h .
  let a u8 be consume h2 .
  let b u8 be consume h .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_twonames.low
mistake_twonames.low:17:0 E-OWN-MOVED: this `owned` value was already MOVED (consumed) —
 using it again is use-after-move, which SPEC-004 §4.8 has always called a compile error
 and which nothing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set <nam
e> <new value>` re-initialises the place — that is how a handle threads through a loop),
```

```
or borrow it LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered yet
(E-IR-UNSUP says so at the call site), so `f (ref h)` is not a way out today
```

A value like `u64` is copied when stored under another name. A value with ownership is not copied; it moves. With two copies nobody could say who destroys it, and destroying both would destroy it twice. After `var h2 owned buffer be h .`, `h2` is the only owner.

A common misconception. Leaving out `drop` leaks the value

examples/ch19/implicit_release.low

```
module implicit_release .
rem run: keep_or_drop 5 1
rem run: keep_or_drop 5 0

type buffer u8 .

rem the path where `c` is false leaves without `drop` --- release cannot fail, so it hap
pens quietly where the lifetime ends
fn keep_or_drop input v u8 . input c u8 . output u8 .
do
  var h owned buffer be v .
  guard eq c 1 . else return 0 .
  drop h .
  return 1 .
end
```

Output

```
$ lowentc --run keep_or_drop implicit_release.low 5 1
keep_or_drop(5, 1) = 1
$ lowentc --run keep_or_drop implicit_release.low 5 0
keep_or_drop(5, 0) = 0
```

The path where `c` is 0 leaves without `drop`, yet it is neither rejected nor leaked. **Release**, which gives memory back, cannot fail, so it happens quietly where the lifetime ends. Only **completion**, which can fail, must be written by the author (`incomplete.low`). Where branches **meet again**, though, the ownership state must match (`join.low`). A path that leaves never meets the other, so that rule does not apply.

19.7 This chapter's syntax at a glance

Shape	Meaning	Why
`input h owned buffer .` · `var h owned buffer be v .`	a value with ownership	one name is responsible for destroying it
`consume h`	passing moves ownership	no use after the move — `E-OWN-MOVED`
`drop h .`	say it is destroyed now	destroying twice is rejected

`set h v .` (after a move)	refill the moved place	a loop body ends in the same shape
different ownership states per branch after `if`	rejected (`E-OWN-JOIN`)	no hidden “was it dropped” flag
`fn finish input j owned journal . output result …`	declares that finishing this type can fail	completion declared without a new word
leaving scope without calling completion	rejected (`E-OWN-INCOMPLETE`)	a failing finish is never swallowed
leaving without a word, when no completion is needed	released quietly	release cannot fail

Table 19.4 — Ownership syntax — shape · meaning · why it looks this way

Recap

`owned t` must be disposed of exactly once and moves when passed. Reusing a moved value, disposing twice, and differing ownership state across branches are rejected. Disposal splits into release, which cannot fail, and completion, which can; a type needing completion is declared by an op that takes it `owned` and returns a `result`. Silently discarding a value that needs completion is rejected. Understand memory rules by strength: static, dynamic and proven.

20 Allocators and fixed memory

In this chapter

20.1 The three parts of an allocator 244
20.2 Cutting borrowed bytes 245
20.3 Swapping allocators 246
20.4 Default allocators that carve straight from a root 249
20.5 Growing and returning a piece — `grow` and `release` 250
20.6 Where the three layers sit in the standard library 252
20.7 Who sets the size of the fixed window? 252
20.8 Same bits, different reading 253
20.9 Common mistakes 254
20.10 This chapter’s syntax at a glance 258

What to know first

chapter 18, Regions · the fixed window and the heap, the two roots
chapter 16, Capabilities · capabilities authorise by kind and cost nothing at run time
chapter 15, Effects · `via` lets a type argument decide allocation effects

Looking back

In chapter 18, what was rejected when building for a machine without an operating system, and what could still be used?

> **A.** The `heap` effect, `cap heap` inputs and `region … heap` blocks were all rejected with `E-HEAP-NOHOST`. `alloc`, which carves from the fixed window that does not grow, could still be used. This chapter covers the **allocators** that hand out space on top of that fixed window, and on top of bytes someone else lent.

The need for this chapter, and its context

A region is a single policy: “all at once when the block ends”. Real programs want more policies. They want to take one big buffer and cut it from the front, to cut with alignment, to run the same code on the heap and on the fixed window. In Lowent an allocator is not a special language device but **an actor that satisfies a trait**. So user-made allocators and standard ones fit the same slot. As the last chapter of Part V on memory, it lays out the path of fixed memory all the way to machines without an operating system.

By the end of this chapter

You will learn that an allocator is made of three parts: a capability, a policy and state. You will use a bump allocator that cuts borrowed bytes, and confirm that running out of memory is a value. You will pick up how to take an allocator as a type parameter and hand it over with `using`, and the default allocators that carve straight from a root, with the rule for capability fields. You will also see how the linker sets the size of the fixed window on machines without an operating system, and `bit_cast`, which keeps the bits and changes only how they are read.

The questions this chapter answers

1 Can a bump allocator take back one piece?

20.1 The three parts of an allocator

Layer	When it exists	What it is
Capability	only at translation	May it touch the root? — `cap allocator` · `cap heap`
Policy	at translation (a type)	By what rule does it carve? — a **type** satisfying the `byte_allocator` trait
State	at run time	The cursor and backing bytes — a **value** of that type (an actor)

Table 20.1 — The three parts of an allocator

The `byte_allocator` trait of the standard library’s `allocs` requires three ops. `reserve n` cuts off `n` bytes and gives an `option`, `grow` enlarges the last piece in place, and `used` answers how much has been used. An actor (chapter 25) is an object that holds its own state and is spoken to only with `send`. Here it is enough to know that it is made with `spawn actor` and called with `send a reserve 3`.

20.2 Cutting borrowed bytes

The simplest allocator is a **bump** allocator. It keeps one cursor, cuts off as much as requested, and pushes the cursor forward.

```
  buf (8 bytes)
  [ a  a  a | b  b | ·  ·  · ]
    └ the piece reserve 3 gave
             └ the piece reserve 2 gave
                   ▲ cursor (used = 5) --- the next reserve cuts from here

asking reserve 4: only 3 bytes are left → none (running short is a value too)
```

examples/ch20/borrowed.low

```
module borrowed .
rem run: carve [0,0,0,0,0,0,0,0]

use allocs .

proc carve input buf mut slice u8 . output u64 . effects state .
do
  var a allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let room u64 be send a init buf .
  let p option mut slice u8 be send a reserve 3 .
  guard is_some p . else return 91 .
  let pv mut slice u8 be some_value p .
  set (index pv 0) 65 .
  let q option mut slice u8 be send a reserve 99 .
  guard not (is_some q) . else return 92 .
  return send a used .
end
```

Output

```
$ lowentc --run carve borrowed.low [0,0,0,0,0,0,0,0]
carve([65,0,0,0,0,0,0,0]) = 3
  arg0 (written) = [65,0,0,0,0,0,0,0]
```

- `spawn actor allocs.bump_bytes` makes the allocator, and `send a init buf` hands it the bytes to cut. This allocator cannot make memory by itself — the discipline of never allocating secretly.
- `send a reserve 3` cuts off 3 bytes. What comes back is not a copy but a slice pointing at part of the original, so `set (index pv 0) 65 .` changes the first byte of the caller's `buf`. The argument `[65,0,…]` the VM shows is the trace.
- `send a reserve 99` gives `none` because there is not enough space. Not a trap. **Running out of memory is a value**, and the caller checks it.

This op's head has neither `cap allocator` nor `alloc`; its effect is only `state`. The caller lent the bytes, and the allocator merely hands them out. The capability draws the line — code without an allocation capability can still allocate fully on bytes someone gave it.

> **Q. Can a bump allocator take back one piece?**
>
> **A.** It takes back only the last piece (`release`, trait `freeing_allocator`). For any other piece it answers `false` and changes nothing. Pretending to accept an unknown piece would let a double

> return erase someone else's space. If the shape is taking and releasing over and over, `pool` with generational handles is the right tool (chapter 35).

20.3 Swapping allocators

Code that uses an allocator need not know which implementation it is. It takes the allocator's **type** as a translation-time parameter and its **value** with a `using` clause.

examples/ch20/generic.low

```
module generic .
rem run: with_bump [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
rem run: with_aligned [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

use allocs .

proc two_from
  input comptime a type .
  using al a .
  output u64 .
  effects state via a .
  requires allocs.byte_allocator a .
do
  let p option mut slice u8 be send al reserve 3 .
  guard is_some p . else return 91 .
  let q option mut slice u8 be send al reserve 5 .
  guard is_some q . else return 92 .
  return send al used .
end

proc with_bump input buf mut slice u8 . output u64 . effects state .
do
  var b allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let c u64 be send b init buf .
  let n u64 using b be two_from .
  return n .
end

proc with_aligned input buf mut slice u8 . output u64 . effects state .
do
  var b allocs.bump_aligned be spawn actor allocs.bump_aligned .
  let c u64 be send b init buf .
  let n u64 using b be two_from .
  return n .
end
```

Output

```
$ lowentc --run with_bump generic.low [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
with_bump([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]) = 8
  arg0 (written) = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
$ lowentc --run with_aligned generic.low [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0]
```

```
with_aligned([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]) = 13
  arg0 (written) = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

- `input comptime a type .` is the allocator's type, fixed to a concrete type at translation (chapter 22).
- `using al a .` receives the allocator value of that type under the name `al`. `using` is not an input. The caller does not write it in an argument position but in the binding, as `let n u64 using b be two_from ..`
- `requires allocs.byte_allocator a .` is the condition that `a` satisfies the trait (chapter 23).
- `effects state via a .` means the effects of the allocator's `reserve` are this op's effects.

Given `bump_bytes`, the same `two_from` uses 3 + 5 = 8; given `bump_aligned`, which aligns start positions to multiples of 8, the second piece starts at 8 and it uses 13. Because the type is fixed at translation, there is no virtual function table and no indirect call. Swapping costs nothing at run time.

If no source is written, a default is chosen: the binding's `using`, the op's own `using` name, and otherwise the input or binding of matching type if there is **only one**. With two or more it does not guess and asks you to write it with `E-ALLOC-AMBIGUOUS`. Defaults do not cross op boundaries — an allocator from the caller never flows in on its own, so **there is no global allocator**.

Order	Source	Why here
1	`using <name>` written on the binding	what you write always wins
2	the **only** name of fitting type among this op's `using` clauses	the allocator the op said it takes
3	the **only** input or binding of fitting type in this op	one visible value leaves nothing to confuse
none	`E-ALLOC-NOSOURCE`	no global allocator fills the gap
two or more	`E-ALLOC-AMBIGUOUS`	no guessing; every candidate is named

Table 20.2 — The order that picks an allocator source — the first match from the top

Fields are not counted. If an allocator hidden inside a struct were used silently, reading the code would not tell you which buffer shrinks.

With no source at all, the call is refused.

examples/ch20/nosource.low

```
module nosource .
rem expect: E-ALLOC-NOSOURCE

use allocs .

proc one_from
  input comptime a type .
  using al a .
  output u64 .
  effects state via a .
  requires allocs.byte_allocator a .
do
  let p option mut slice u8 be send al reserve 3 .
  guard is_some p . else return 91 .
```

```
  return send al used .
end

proc caller output u64 . effects state .
do
  rem ✘ there is no allocator at all in this op to draw from
  let n u64 be one_from .
  return n .
end
```

Output

```
$ lowentc --check nosource.low
21:16 E-ALLOC-NOSOURCE: this call draws from an allocator, but NO value of a fitting typ
e is visible in this op — no input, no binding, no `using` clause. There is no global al
locator: take one as an input, create one here, or declare `using <name> <type> .` on th
is op (RFC-0112 D8(4))
nosource.low:19:1 W-EFFECT-OVER: this op DECLARES `state` but never PERFORMS it. A decla
red effect is a cost the caller must budget for (a pure caller cannot call an `io` op).
Drop it, or — if a future version will perform it — say so (RFC-0007)
```

`caller` has no input, no spawned actor and no `using` clause. In another language a global heap would quietly step in here. Lowent asks you to take an allocator as an input, create one here, or write a `using` clause. Because the call is refused, `caller` never really uses `state` either, so `W-EFFECT-OVER` comes along. It goes away once the first error is fixed.

The other way round, writing `using` on a call that draws from no allocator is refused too.

examples/ch20/usingunused.low

```
module usingunused .
rem expect: E-ALLOC-USING-UNUSED

use allocs .

fn twice input x u64 . output u64 . do
  return mul x 2 .
end

proc caller input buf mut slice u8 . output u64 . effects state .
do
  var b allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let c u64 be send b init buf .
  rem ✘ twice draws from no allocator, yet the binding picks one with using
  let n u64 using b be twice 4 .
  return n .
end
```

Output

```
$ lowentc --check usingunused.low
15:19 E-ALLOC-USING-UNUSED: this binding says which allocator to use, but the call it in
itialises does not draw from one (no `using` clause on that op). An object that already
carries its allocator — like `vecgen.append` on a vector — does not take the caller's ch
oice (RFC-0112 D8(5))
```

`twice` only doubles a number and takes no allocator. A choice that will never be used misleads the reader into thinking `twice` uses memory. So whatever you write must be used.

20.4 Default allocators that carve straight from a root

`allocs` also provides two allocators that carve straight from a root, under the same trait.

Actor	State	Effect of `reserve`
`fixed_bytes`	`root cap allocator .` · amount used	`alloc state` — anywhere
`heap_bytes`	`root cap heap .` · amount used	`heap state` — only with an operating system

Table 20.3 — Default allocators

examples/ch20/fixed.low

```
module fixed .
rem run: main

use allocs .

proc main input al cap allocator . output u8 . effects alloc state .
do
  var fb allocs.fixed_bytes be spawn actor allocs.fixed_bytes .
  let a option mut slice u8 be send fb reserve 64 .
  guard is_some a . else return 1 .
  let b option mut slice u8 be send fb reserve 64 .
  guard is_some b . else return 2 .
  return narrow u8 (send fb used) .
end
```

Output

```
$ lowentc --run main fixed.low
main() = 128
```

`fixed_bytes`'s state has a **capability field**. A capability is not a run-time value, so the field has no size. Instead a rule comes with it: an actor with a capability field can be spawned only **from an op that holds a capability of the same kind**.

examples/ch20/forge.low

```
module forge .
rem expect: E-CAP-FORGE

use allocs .

proc sneaky output u64 . effects heap state .
do
  var hb allocs.heap_bytes be spawn actor allocs.heap_bytes .
  return send hb used .
end
```

Output

```
$ lowentc --check forge.low
forge.low:8:0 E-CAP-FORGE: this spawns an actor whose state HOLDS a capability (`cap all
ocator` / `cap heap`), but the op spawning it holds no capability of that kind. A capabi
lity field means nothing at run time — it is only real because the place that creates th
e actor already had the right. Without that, one `spawn` would forge authority out of th
in air (RFC-0112 D6). Take the capability as an input of this op
forge.low:7:1 W-EFFECT-OVER: this op DECLARES `heap` but never PERFORMS it. A declared e
ffect is a cost the caller must budget for (a pure caller cannot call an `io` op). Drop
it, or — if a future version will perform it — say so (RFC-0007)
```

`sneaky` tried to spawn `heap_bytes` without receiving `cap heap`. If that were allowed, a heap could be conjured in one line where there is no capability. Capabilities are handed over, not picked up.

Receive the capability and the same thing works.

examples/ch20/heapbytes.low

```
module heapbytes .
rem run: main

use allocs .

proc main input h cap heap . output u8 . effects heap state .
do
  var hb allocs.heap_bytes be spawn actor allocs.heap_bytes .
  rem the heap hands out 100000 bytes, more than the fixed window (64 KiB by default) ho
lds
  let a option mut slice u8 be send hb reserve 100000 .
  guard is_some a . else return 1 .
  rem return 2 if the amount used is not 100000, and 0 if it is
  guard eq (send hb used) 100000 . else return 2 .
  return 0 .
end
```

Output

```
$ lowentc --run main heapbytes.low
main() = 0
```

`main` receives `cap heap` and writes `heap` in its effects line. So it may spawn `heap_bytes`, and it gets 100000 bytes, more than the fixed window holds. When the heap runs short it chains another chunk. Built for a machine without an operating system, this file is refused with `E-HEAP-NOHOST` (chapter 18).

20.5 Growing and returning a piece — `grow` and `release`

A bump allocator only moves forward. Even so, **the piece it handed out last** is safe to take back, because nobody has received a place after it. `grow` enlarges that piece in place, and `release` (trait `freeing_allocator`) takes it back.

examples/ch20/growrelease.low

```
module growrelease .
rem run: regrow [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
use allocs .

proc regrow input buf mut slice u8 . output u64 . effects state .
do
  var b allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let c u64 be send b init buf .
  let p option mut slice u8 be send b reserve 4 .
  guard is_some p . else return 91 .
  let pv mut slice u8 be some_value p .
  rem grow the piece just received from 4 to 6 bytes in place
  let g option mut slice u8 be send b grow pv 6 .
  guard is_some g . else return 92 .
  let gv mut slice u8 be some_value g .
  let q option mut slice u8 be send b reserve 2 .
  guard is_some q . else return 93 .
  let qv mut slice u8 be some_value q .
  rem gv is not the last piece (qv comes after it), so it is not taken back
  let back1 bool be send b release gv .
  rem qv is the last piece, so it is taken back
  let back2 bool be send b release qv .
  let u u64 be send b used .
  var d1 u64 be 0 .
  if back1 . do
    set d1 1 .
  end
  var d2 u64 be 0 .
  if back2 . do
    set d2 1 .
  end
  rem pack the amount used, the first answer and the second answer into one three-digit
number
  return add (mul u 100) (add (mul d1 10) d2) .
end
```

Output

```
$ lowentc --run regrow growrelease.low [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
regrow([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]) = 601
  arg0 (written) = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

- `send b grow pv 6` grows `pv` from 4 bytes to 6. What it grows is **the piece itself**, not a size. The implementation checks with the builtin `same_slice a b` (same start address and same length?) that `pv` is exactly the bytes it just handed out. Pass someone else's buffer of the same length and the answer is `none`. Recognising a piece by size alone would let two containers overlap without a sound.
- After `qv` is handed out, `gv` is no longer the last piece. So `release gv` is `false` and changes nothing.
- `release qv` is `true`. The cursor goes back to 6, so `used` is 6. The answer 601 reads "used 6 · first answer false · second answer true".

Both ops are **an optimisation, not a promise**. If a piece cannot grow, the caller receives a new one and copies, and the answer must be the same. For `fixed_bytes` and `heap_bytes`, which carve straight from a root, `grow` is always `none`. A root does not know whose piece came last.

20.6 Where the three layers sit in the standard library

Module · name	Capability	Policy (type)	State (value)
allocs.bump_bytes · bump_aligned	none — borrowed bytes	an actor with byte_allocator · freeing_allocator	backing bytes · cursor
allocs.fixed_bytes · heap_bytes	capability field cap allocator · cap heap	byte_allocator	amount used
vecgen.vec t a	none	takes the allocator type a as a parameter	element count · buffer — open takes the allocator with using
growvec.gvec	none	a name fixed to vecgen.vec u8 allocs.bump_bytes	same as vecgen
pool.block_pool b	none — borrowed bytes	not an allocator — a struct that takes blocks back one by one through generation handles	blocks · generation array

Table 20.4 — Allocation-related modules — which of capability, policy and state each one carries

Only the two actors that reach a root hold a capability. Everything else works on bytes someone handed over. So library code that received no capability can still build and grow containers, and to find the code that brings new memory into the program you only look at the ops that received a capability.

20.7 Who sets the size of the fixed window?

On a machine without an operating system, the fixed window is the memory between two symbols set by the linker. Its size is not baked into the executable. The compiler emits a linker-script fragment, and only the size in that fragment changes per board.

```
$ lowentc --emit-ldscript --fixed-bytes 4096 fixed.low
.lw_fixed (NOLOAD) : ALIGN(8)
{
    __lw_fixed_start = .;
    . = . + 4096;
    __lw_fixed_end = .;
}
```

To imitate another board on the host, give the VM a window size. `fixed.low` asks for 64 bytes twice, so shrinking the window to 100 bytes with `lowentc --fixed-bytes 100 --run main fixed.low` makes the second `reserve` return `none` and prints `main() = 2`. You can see on your development machine where a small machine runs out of memory.

> **A common misconception. Embedded code does not allocate dynamically, so it needs no allocator**
>
> The real meaning of “no dynamic allocation” is not to use **a growing heap** and **allocation that hides its failures**. Carving from a window fixed once at boot, returning shortage as a value, and rewinding all at once do not conflict with that discipline. Lowent’s fixed window has its size written by the linker, shortage is `none`, and the heap effect is blocked at translation. The grammar keeps the rule.

20.8 Same bits, different reading

Reading the same bytes as a different type comes in two kinds. `view`, which lays a struct layout over a byte slice, was seen in chapter 13. Keeping the bit pattern between two scalars of the same width and changing only how it is read is `bit_cast`.

examples/ch20/bitcast.low

```
module bitcast .
rem run: reinterpret -1
rem run: float_bits 1.0

fn reinterpret input x i32 . output u32 .
do
  return bit_cast u32 x .
end

fn float_bits input f f64 . output u64 .
do
  return bit_cast u64 f .
end
```

Output

```
$ lowentc --run reinterpret bitcast.low -1
reinterpret(-1) = 4294967295
$ lowentc --run float_bits bitcast.low 1.0
float_bits(1.0) = 4607182418800017408
```

The −1 of an `i32` has every bit set, so read as a `u32` it is 4294967295. Reading the `f64` 1.0 as a `u64` shows its IEEE 754 representation as is. Unlike `cast`, it does not move the value, so it never stops.

The target type must be one where **every bit pattern is a valid value**. `bool` and `enum` are not.

examples/ch20/bitcast_bool.low

```
module bitcast_bool .
rem expect: E-TYPE-BITCAST

fn truthy input a u8 . output bool .
do
  return bit_cast bool a .
end
```

Output

```
$ lowentc --check bitcast_bool.low
bitcast_bool.low:6:0 E-TYPE-BITCAST: bit_cast's target must be a `plain` scalar type — e
very bit pattern valid, no padding (SPEC-004 §190). bool/enum have trap representations
and are NOT plain, so they are outside safe punning
```

Reading the `u8` value 2 as a `bool` would give a value that is neither true nor false. The path that creates such values is closed.

20.9 Common mistakes

Counter-example. Forgetting `init` on a bump allocator

examples/ch20/mistake_noinit.low

```
module mistake_noinit .
rem expect: E-ACTOR-UNINIT

use allocs .

proc carve input buf mut slice u8 . output u64 . effects state .
do
  var a allocs.bump_bytes be spawn actor allocs.bump_bytes .
  rem ✘ `send a init buf` is missing --- asks an allocator that has no bytes to hand out
  let p option mut slice u8 be send a reserve 3 .
  guard is_some p . else return 91 .
  return send a used .
end
```

Output

```
$ lowentc --check mistake_noinit.low
mistake_noinit.low:10:0 E-ACTOR-UNINIT: this is the FIRST message sent to an actor that
was just spawned, and the handler READS a state field that holds a slice without setting
 it. A fresh actor's state is all zeroes, and zero is not a slice: the VM stops (`len ne
eds a slice`) while native quietly answers `none` — the two back ends disagree, which me
ans one of them is lying. Send the message that sets it up first (the one whose handler
`set`s that field, typically `init`)
```

`bump_bytes` does not create memory by itself. Until bytes are attached with `send a init buf`, it has nothing to hand out. A just-spawned actor's state fields are all zeroes, and zero is not a slice — reading that field makes the VM and the native build run differently. So translation refuses it (`E-ACTOR-UNINIT`). Put the `init` on the line right after spawning the allocator.

Counter-example. Attaching the same bytes to two allocators

examples/ch20/mistake_sharedbuf.low

```
module mistake_sharedbuf .
rem expect: E-EXCL

use allocs .
```

```
proc twice input buf mut slice u8 . output u8 . effects state .
do
  var a allocs.bump_bytes be spawn actor allocs.bump_bytes .
  var b allocs.bump_bytes be spawn actor allocs.bump_bytes .
  rem ✘ the same bytes are given to two allocators --- both hand out the same place
  let c1 u64 be send a init buf .
  let c2 u64 be send b init buf .
  let p option mut slice u8 be send a reserve 3 .
  let q option mut slice u8 be send b reserve 3 .
  guard is_some p . else return 91 .
  guard is_some q . else return 92 .
  let pv mut slice u8 be some_value p .
  let qv mut slice u8 be some_value q .
  set (index pv 0) 65 .
  set (index qv 0) 66 .
  rem 65 was written, yet 66 comes back
  return index pv 0 .
end
```

Output

```
$ lowentc --check mistake_sharedbuf.low
mistake_sharedbuf.low:12:0 E-EXCL: the same WRITABLE place was handed to a SECOND actor.
 A write borrow is exclusive (§8.4): two actors holding the same bytes both hand them ou
t, so two containers silently overlap and a write through one is read through the other
(measured: 65 written, 66 read back). Give each actor its own bytes — `subslice` the buf
fer into pieces that do not overlap
```

The two allocators know nothing of each other. Both cut from the front of `buf`, so `pv` and `qv` become the same place: write 65 and then 66, and reading `pv` gives 66. The rule that there is only one write borrow (chapter 12) must hold across the actor boundary too, so this is refused with `E-EXCL` (until 2026-09-16 it passed). Attach separate bytes to each allocator; if one buffer must be shared out, cut two non-overlapping pieces with `subslice`.

Counter-example. Leaving out `using` when two allocators fit

examples/ch20/mistake_ambiguous.low

```
module mistake_ambiguous .
rem expect: E-ALLOC-AMBIGUOUS

use allocs .

proc two_from
  input comptime a type .
  using al a .
  output u64 .
  effects state via a .
  requires allocs.byte_allocator a .
do
  let p option mut slice u8 be send al reserve 3 .
  guard is_some p . else return 91 .
  return send al used .
```

```
end

proc with_two input small mut slice u8 . input big mut slice u8 . output u64 . effects s
tate .
do
  var s allocs.bump_bytes be spawn actor allocs.bump_bytes .
  var g allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let x u64 be send s init small .
  let y u64 be send g init big .
  rem ✘ two allocators of a fitting type are in scope, and the call does not say which
  let n u64 be two_from .
  return n .
end
```

Output

```
$ lowentc --check mistake_ambiguous.low
25:16 E-ALLOC-AMBIGUOUS: this call draws from an allocator, and MORE THAN ONE value of a
 fitting type is in scope — the tool will not guess which one you meant. Say it on the b
inding: `let <name> <type> using <allocator> be …` (RFC-0112 D8(4))
```

`s` and `g` are both `bump_bytes`, so the tool cannot guess. A guess might carve from the big buffer what should come from the small one, or the other way round, and such a bug stays hidden on a development machine with plenty of memory. So it stops with `E-ALLOC-AMBIGUOUS` and asks you to say which.

examples/ch20/ambiguous_fixed.low

```
module ambiguous_fixed .
rem run: with_two [0,0,0,0] [0,0,0,0,0,0,0,0]

use allocs .

proc two_from
  input comptime a type .
  using al a .
  output u64 .
  effects state via a .
  requires allocs.byte_allocator a .
do
  let p option mut slice u8 be send al reserve 3 .
  guard is_some p . else return 91 .
  return send al used .
end

proc with_two input small mut slice u8 . input big mut slice u8 . output u64 . effects s
tate .
do
  var s allocs.bump_bytes be spawn actor allocs.bump_bytes .
  var g allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let x u64 be send s init small .
  let y u64 be send g init big .
  rem name the source on the binding with `using`
  let n u64 using g be two_from .
  let m u64 using g be two_from .
```

```
  return m .
end
```

Output

```
$ lowentc --run with_two ambiguous_fixed.low [0,0,0,0] [0,0,0,0,0,0,0,0]
with_two([0,0,0,0], [0,0,0,0,0,0,0,0]) = 6
  arg0 (written) = [0,0,0,0]
  arg1 (written) = [0,0,0,0,0,0,0,0]
```

The two calls to `two_from` each carved 3 bytes from the same `g`, so `used` is 6. With the source written on every call, you can read which buffer shrinks.

Counter-example. Leaving out `via a` in a generic op

examples/ch20/mistake_novia.low

```
module mistake_novia .
rem expect: E-EFFECT

use allocs .

proc one_from
  input comptime a type .
  using al a .
  output u64 .
  rem ✘ no `via a` --- the `alloc` the allocator performs is not part of this op's decla
ration
  effects state .
  requires allocs.byte_allocator a .
do
  let p option mut slice u8 be send al reserve 3 .
  guard is_some p . else return 91 .
  return send al used .
end

proc with_fixed input al cap allocator . output u64 . effects alloc state .
do
  var fb allocs.fixed_bytes be spawn actor allocs.fixed_bytes .
  let n u64 using fb be one_from .
  return n .
end
```

Output

```
$ lowentc --check mistake_novia.low
mistake_novia.low:13:1 E-EFFECT: this op performs `alloc`, which its `effects` clause do
es not declare — add it to `effects …`, or stop calling what needs it
mistake_novia.low:20:1 W-EFFECT-OVER: this op DECLARES `alloc` but never PERFORMS it. A
declared effect is a cost the caller must budget for (a pure caller cannot call an `io`
op). Drop it, or — if a future version will perform it — say so (RFC-0007)
```

`one_from` declares only `effects state .`, but when instantiated with `fixed_bytes`, `reserve` performs `alloc`. `via a` is what makes “the allocation effects of type `a` are part of my declaration”

true, so each instance gets exact effects. Without it you get `E-EFFECT`, and since the effect does not reach the caller, `with_fixed` oddly gets a `W-EFFECT-OVER` as well. Fixing the first error removes the second.

A common misconception. Spawning another allocator gives you another window

examples/ch20/shared_window.low

```
module shared_window .
rem run: main

use allocs .

proc main input al cap allocator . output u8 . effects alloc state .
do
  rem two allocators, but only one fixed window
  var f1 allocs.fixed_bytes be spawn actor allocs.fixed_bytes .
  var f2 allocs.fixed_bytes be spawn actor allocs.fixed_bytes .
  let a option mut slice u8 be send f1 reserve 40000 .
  guard is_some a . else return 1 .
  rem `f2` has used nothing yet, but the window has too little room left
  let b option mut slice u8 be send f2 reserve 40000 .
  guard is_some b . else return 2 .
  return 0 .
end
```

Output

```
$ lowentc --run main shared_window.low
main() = 2
```

`fixed_bytes` carves from the **root**. There is one root and one cursor (chapter 18). Once `f1` takes 40000 bytes, `f2` gets `none` even though it has used nothing yet, because the default window (65536 bytes) has too little room left. An allocator's `used` is the amount **that allocator** has used, not what is left in the whole window. If you need separate budgets, take one large piece from the window and attach non-overlapping parts of it to several `bump_bytes`.

20.10 This chapter's syntax at a glance

Shape	Meaning	Why
`var a allocs.bump_bytes be spawn actor allocs.bump_bytes .`	spawn an allocator (its state)	state is an actor value — there is no global allocator
`send a init buf`	attach the bytes to hand out	an allocator never creates memory behind your back
`send a reserve 3 · send a used`	request a piece (`option`) · amount used	shortage is a value, not a trap
`input comptime a type .`	receive the allocator's type (policy) at translation time	swapping costs nothing at run time

`using al a .`	receive an allocator value of that type — not an input	it does not sit among the call's arguments
`let n u64 using g be two_from .`	say which allocator this call carves from	with two or more, nothing is guessed
`effects state via a . · requires allocs.byte_allocator a .`	inherit the type's effects · trait condition	exact effects per instance
`allocs.fixed_bytes` · `allocs.heap_bytes`	default allocators carving straight from a root	only an op holding that kind of capability may spawn one — `E-CAP-FORGE`
`send b grow pv 6 · send b release qv`	grows the last piece · takes it back	checks identity with `same_slice`, not size
no source · an unused `using`	`E-ALLOC-NOSOURCE` · `E-ALLOC-USING-UNUSED`	no global allocator, and no empty choice
`bit_cast u32 x`	keep the bits, change only how they are read	never read as `bool` or `enum`

Table 20.5 — Allocator syntax — shape · meaning · why it looks this way

Recap

An allocator is a capability, a policy (a type satisfying a trait) and state (an actor value). A bump allocator cuts borrowed bytes and reports shortage as `none`. Allocators are received with `input comptime a type .` and `using al a .`, swapped at no run-time cost, and there is no global allocator. `fixed_bytes` and `heap_bytes` are default allocators with capability fields and can be spawned only from ops holding the same kind of capability. The linker sets the size of the fixed window. `bit_cast` keeps the bits and never reads them as `bool` or `enum`.

Part VI — Abstraction

21 Modules — hidden by default

In this chapter

21.1 One file, one module 262
21.2 Importing 263
21.3 What can be exported, and how to shorten a name 264
21.4 There is no search path 265
21.5 When names collide 266
21.6 Top-level order does not matter 266
21.7 Common mistakes 267
21.8 This chapter's syntax at a glance 272

What to know first

chapter 3, The surface · a dotted name is a path to a declared name
chapter 16, Capabilities · the heads of exported ops show what a library touches

Looking back

In chapter 16's last case study, what did you need to look at to find out whether an imported library reaches the network?

> **A.** You look through the heads of the ops the library **exports** for any that take `cap net` (and `cap c` or `cap machine`). This chapter covers what "export" means, and how to bring in names from other files.

The need for this chapter, and its context

Once a program is more than one file, two questions come up: **what is visible from outside?** and **what does this file depend on?** If the first answer is blurry, you cannot tell who breaks when internals change; if the second is blurry, the same source brings in different things on different machines. Lowent has both written in the source. As the first chapter of Part VI, "Abstraction", it starts from the outermost unit of splitting code.

By the end of this chapter

You will learn that one file is one module and that only things marked `export` are visible outside. You will pick up how to import another file with `use … from "place"`, how to pass several files as one compilation unit, and how to import standard-library names. You will also see that qualifying with the module name does not reach hidden names, the design decision to have no search path, and that the order of top-level declarations does not matter.

The questions this chapter answers

1 Where does `use geom .`, with no `from "…"`, look?

21.1 One file, one module

One source file is one module. A module writes its name on the first line, and the module name need not match the file name.

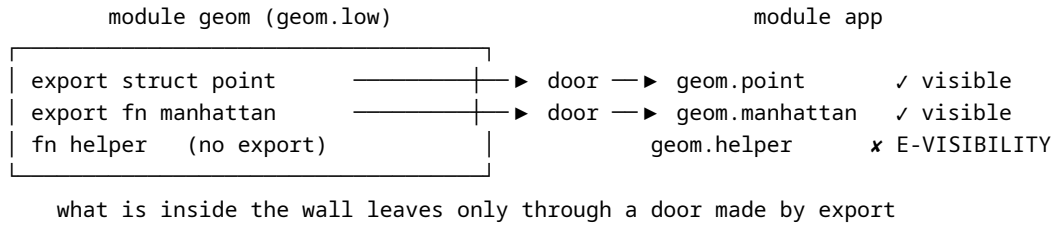
examples/ch21/geom.low

```
module geom .

export struct point do
  x i64 .
  y i64 .
end

export fn manhattan input a point . input b point . output i64 .
  requires ge (field a x) -1000000 .
  requires le (field a x) 1000000 .
  requires ge (field b x) -1000000 .
  requires le (field b x) 1000000 .
  requires ge (field a y) -1000000 .
  requires le (field a y) 1000000 .
  requires ge (field b y) -1000000 .
  requires le (field b y) 1000000 .
do
  return add (abs (sub (field b x) (field a x))) (abs (sub (field b y) (field a y))) .
end

fn helper input v i64 . output i64 .
do
  return v .
end
```

Output

```
$ lowentc --check geom.low
== check: ok ==
```

The `geom` module exports the `point` struct and the `manhattan` op with `export`. `helper` has no `export`, so it can be used only inside this module.



Hidden is the default. What is visible from outside is a promise, and hiding something by mistake is easier to fix than promising something by mistake. When emitting native code, too, only `export`ed ops become symbols callable from C; everything else is `static`.

21.2 Importing

`use <module> from "<place>" .` imports a module from the file at that place. The place is a path **relative to the file that declares it**.

examples/ch21/app.low

```
module app .
rem run: distance 3 4

use geom from "geom.low" .

fn distance input dx i64 . input dy i64 . output i64 .
  requires ge dx -1000 .
  requires le dx 1000 .
  requires ge dy -1000 .
  requires le dy 1000 .
do
  let a geom.point be make geom.point do x 0 . y 0 . end .
  let b geom.point be make geom.point do x dx . y dy . end .
  return geom.manhattan a b .
end
```

Output

```
$ lowentc --run distance app.low 3 4
distance(3, 4) = 7
```

Imported names are called qualified by the module name — `geom.point`, `geom.manhattan`. The attached dot is not an operation on a value but a path to a declared name (chapter 3).

Qualification does not reach hidden names.

examples/ch21/hidden.low

```
module hidden .
rem expect: E-VISIBILITY

use geom from "geom.low" .

fn peek input v i64 . output i64 .
do
  return geom.helper v .
end
```

Output

```
$ lowentc --check hidden.low
hidden.low:8:0 E-VISIBILITY: this name belongs to ANOTHER module and is not `export`ed —
 qualifying it does not open the door. A module that cannot keep anything private is not
 a module, it is a prefix. Mark it `export`, or stop reaching in (RFC-0011 §6.2)
```

In the words of the diagnostic, “a module that cannot keep anything private is not a module, it is a prefix”.

> **Q. Where does `use geom .`, with no `from "…"`, look?**
>
> **A.** Inside the same compilation unit. Passing several files, as in `lowentc --check app.low geom.low`, makes those files one unit. If the module is not in the unit, the compiler says with `W-USE-EXTERNAL` that it cannot confirm the module exists, and rejects actual uses of the name. Standard-library names (`use allocs .`) resolve from where the standard modules are installed.

21.3 What can be exported, and how to shorten a name

`export` goes on most named declarations. Some do not take it.

Declaration	`export`	Why
`fn` · `proc`	yes	ops other modules call
`struct` · `enum` · `type` · `newtype`	yes	the input and output types of an exported op must be exported too, or it cannot be used
`trait` · `actor`	yes	a promise types in other modules satisfy · an actor other modules spawn
top-level `let` (constant)	no — `E-TOPLEVEL`	constants stay inside the module; to share one, export a `fn` that returns the value
`test`	no — `E-TOPLEVEL`	tests belong to whoever builds that module

Table 21.1 — Declarations that take `export`

When a module name is long, or two modules share a name, shorten it with `as`.

examples/ch21/alias.low

```
module alias .
rem run: distance 3 4

rem call the imported module by the short name g
use geom from "geom.low" as g .

fn distance input dx i64 . input dy i64 . output i64 .
  requires ge dx -1000 .
  requires le dx 1000 .
  requires ge dy -1000 .
  requires le dy 1000 .
do
  let a g.point be make g.point do x 0 . y 0 . end .
  let b g.point be make g.point do x dx . y dy . end .
  return g.manhattan a b .
end
```

Output

```
$ lowentc --run distance alias.low 3 4
distance(3, 4) = 7
```

After `use geom from "geom.low" as g .` you call `g.point` and `g.manhattan`. The alias is a name used only inside this file; the module's real name (`geom`) does not change. Inside this file, though, the original name no longer stands — writing `geom.point` is `E-USE-ALIASED`. An alias is a rename,

not a second name: if a module could be called by two spellings, a reader would have to check that they mean the same thing. Importing the same name twice gives `E-NAME-COLLISION`, and the remedy that diagnostic suggests is this alias.

> **Counter-example. Writing an alias and then using the original name**

examples/ch21/mistake_aliasold.low

```
module mistake_aliasold .
rem expect: E-USE-ALIASED

use geom from "geom.low" as g .

fn origin_x output i64 .
do
  rem ✘ an alias was written, yet the original name is used --- the same module would be
 called by two spellings
  let p geom.point be make geom.point do x 7 . y 0 . end .
  return field p x .
end
```

Output

```
$ lowentc --check mistake_aliasold.low
mistake_aliasold.low:9:9 E-USE-ALIASED: this unit renamed that module with `as`, so its
name here is `g` — the original name no longer stands. One thing, one spelling: if both
names worked, a reader would have to check that they mean the same module. Write the mem
ber under `g.`
mistake_aliasold.low:9:28 E-USE-ALIASED: this unit renamed that module with `as`, so its
 name here is `g` — the original name no longer stands. One thing, one spelling: if both
 names worked, a reader would have to check that they mean the same module. Write the me
mber under `g.`
```

The moment you write `as g`, the module's name in this file is `g` and nothing else. `geom.point` is refused with `E-USE-ALIASED`. If both spellings stood, code would get changed on one side only, and a reader would have to keep checking that the two are the same module. The one place the original name belongs is the `use` line.

21.4 There is no search path

Many languages fetch a module from somewhere on a search path when you write just its name. Then what the program depends on becomes knowledge outside the source, and the same source can bring in different files on different machines. Importing in Lowent is one of three things.

- Write the place: `use geom from "geom.low" .`
- Pass it in the same compilation unit: `lowentc --check app.low geom.low`
- Use a reserved standard-library name: `use allocs .`

The principle is the same when a package manifest (`pkg.low`) lists external dependencies. Dependencies are pinned **with their content hash**, and if the bytes change the build is refused (chapter 31).

> **A common misconception. A module's name comes from its file name**
>
> This trips people up in the standard library. What `use` looks for is the `module` declaration inside the file. The module name of `lib/alloc.low` is `allocs`, `lib/str.low` is `strings`, and `lib/vec.low` is `vecs`. The list of modules whose names differ is the first table of the standard-library part (chapter 32).

21.5 When names collide

A module cannot declare the same name twice.

examples/ch21/dup.low

```
module dup .
rem expect: E-NAME-DUP

fn answer output u32 . do return 1 . end
fn answer output u32 . do return 2 . end
```

Output

```
$ lowentc --check dup.low
dup.low:5:4 E-NAME-DUP: `answer` (op) is declared twice in module `dup` — a qualifier ca
nnot tell the two apart (both are `<module>.<name>`), so the second does not hide the fi
rst: it silently makes one of them unreachable
```

The second does not hide the first; rather, one of the two silently becomes unreachable, so it is rejected. When imported names collide, the call site names the module explicitly. Importing never overwrites a name that already exists (`E-NAME-COLLISION`).

21.6 Top-level order does not matter

Top-level declarations in a module are independent of order. An op declared later can be called earlier, and ops can call each other.

examples/ch21/order.low

```
module order .
rem run: is_even 10
rem run: is_even 7

fn is_even input n u32 . output bool .
  requires le n 1000 .
do
  if eq n 0 . do return true . end
  return is_odd (sub n 1) .
end

fn is_odd input n u32 . output bool .
  requires le n 1000 .
do
  if eq n 0 . do return false . end
```

```
  return is_even (sub n 1) .
end
```

Output

```
$ lowentc --run is_even order.low 10
is_even(10) = 1
$ lowentc --run is_even order.low 7
is_even(7) = 0
```

`is_even` calls `is_odd`, declared after it, and `is_odd` calls `is_even` back. Local names **inside an op body**, on the other hand, must be declared before use. A file as a whole is skimmed, but a body is read top to bottom.

21.7 Common mistakes

Counter-example. Using an imported name without its module name

examples/ch21/mistake_bare.low

```
module mistake_bare .
rem expect: E-VISIBILITY

use geom from "geom.low" .

fn origin_x output i64 .
do
  rem ✘ an imported name is used without its module name
  let a point be make point do x 0 . y 0 . end .
  return field a x .
end
```

Output

```
$ lowentc --check mistake_bare.low
mistake_bare.low:9:0 E-VISIBILITY: this name is another module's export, reached BARE —
a cross-module name must be QUALIFIED: write `<module>.<name>` (RFC-0011 qualified-by-de
fault; there is no glob import)
mistake_bare.low:9:0 E-VISIBILITY: this name is another module's export, reached BARE —
a cross-module name must be QUALIFIED: write `<module>.<name>` (RFC-0011 qualified-by-de
fault; there is no glob import)
```

Some languages let you use imported names bare, or spill them all with `import *`. Then you cannot tell by reading whether `point` belongs to this file or to some module, and names collide quietly as imports grow. Lowent has no spilling: a name from another module is always qualified, as in `geom.point`. The diagnostic appears twice on the same line because the type position of `let` and the `make` position are counted separately.

Counter-example. Leaving the extension out of the `from` place

examples/ch21/mistake_noext.low

```
module mistake_noext .
rem expect: E-DEP-MISSING

rem ✘ the place is the exact file name --- include `.low`
use geom from "geom" .

fn same input v i64 . output i64 .
do
  return v .
end
```

Output

```
$ lowentc --check mistake_noext.low
E-DEP-MISSING: `use geom from "geom"` — cannot read that source. A dependency the tool c
annot see is a dependency nobody checked
```

The place is never guessed. Write `"geom"` and the tool looks for a file named exactly `geom`; if there is none, it is `E-DEP-MISSING`. The reason is the same as for having no search path: once the tool starts trying `.low` or searching other directories, what the program depends on moves outside the source again. Write the file name exactly, as in `use geom from "geom.low" .`.

Counter-example. Importing a standard module by its file name

examples/ch21/mistake_filename.low

```
module mistake_filename .
rem expect: E-IR-UNDEF

rem ✘ the file is `lib/str.low`, but the module inside it is named `strings`
use str .

fn same input a slice u8 . input b slice u8 . output bool .
do
  return str.eq_str a b .
end
```

Output

```
$ lowentc --check mistake_filename.low
mistake_filename.low:5:0 W-USE-EXTERNAL: this module is not in the compilation unit — th
ere is no module search path, so nothing here can confirm it exists. Pass the file that
declares it and it WILL be checked (several .low files link into one unit)
mistake_filename.low:9:0 E-IR-UNDEF: undefined name `str.eq_str` — `str` is not a module
 used by this unit (`use str .`) nor an enum
```

The module in the file `lib/str.low` is named `strings`. `use str .` finds no such module, so it warns with `W-USE-EXTERNAL` (“cannot confirm it exists”), and the place that actually uses the name is rejected with `E-IR-UNDEF`. Read the warning first and check the module name.

examples/ch21/filename_fixed.low

```
module filename_fixed .
rem run: same [104,105] [104,105]
```

```
rem run: same [104,105] [104,111]

use strings .

fn same input a slice u8 . input b slice u8 . output bool .
do
  return strings.eq_str a b .
end
```

Output

```
$ lowentc --run same filename_fixed.low [104,105] [104,105]
same([104,105], [104,105]) = 1
  arg0 (written) = [104,105]
  arg1 (written) = [104,105]
$ lowentc --run same filename_fixed.low [104,105] [104,111]
same([104,105], [104,111]) = 0
  arg0 (written) = [104,105]
  arg1 (written) = [104,111]
```

Counter-example. Using a hidden type in an exported op's signature

examples/ch21/secretbox.low

```
module secretbox .
rem expect: W-EXPORT-HIDDEN

rem a type without `export` --- its name cannot be written outside this module
struct secret do
  v u64 .
end

rem ✘ an exported op returns the hidden type --- the importing side cannot write that na
me
export fn make_secret output secret .
do
  return make secret do v 1 . end .
end
```

Output

```
$ lowentc --check secretbox.low
secretbox.low:10:0 W-EXPORT-HIDDEN: this op is exported and its signature names a type t
his module keeps to itself. An importing module cannot write that type, so it has nowher
e to put the value and the export cannot be used from outside (§6.10.1). Export the type
 too, or give the op a signature made of types the other side can name
```

`secretbox` itself translates, but the exporting side is warned with `W-EXPORT-HIDDEN`: this export cannot be used from outside. Without that warning the problem only shows up in the module that imports it.

examples/ch21/mistake_privtype.low

```
module mistake_privtype .
rem expect: E-VISIBILITY

use secretbox from "secretbox.low" .

fn peek output u64 .
do
  rem ✘ the type name needed to hold the result is hidden
  let s secretbox.secret be secretbox.make_secret .
  return field s v .
end
```

Output

```
$ lowentc --check mistake_privtype.low
mistake_privtype.low:9:0 E-VISIBILITY: this name belongs to ANOTHER module and is not `e
xport`ed — qualifying it does not open the door. A module that cannot keep anything priv
ate is not a module, it is a prefix. Mark it `export`, or stop reaching in (RFC-0011 §6.
2)
secretbox.low:10:0 W-EXPORT-HIDDEN: this op is exported and its signature names a type t
his module keeps to itself. An importing module cannot write that type, so it has nowher
e to put the value and the export cannot be used from outside (§6.10.1). Export the type
 too, or give the op a signature made of types the other side can name
```

`make_secret` is exported, but the type of its result, `secret`, is hidden. The importer cannot write a name to hold the result and is rejected with `E-VISIBILITY`. The export is unusable. Export the input and output types of an exported op as well.

Counter-example. Writing another module's enum variant with the module name in a `case`

examples/ch21/sizes.low

```
module sizes .

rem an enum for other modules to use; its variants carry no value
export enum kind do
  small .
  big .
end

export fn classify input n u64 . output kind . do
  if lt n 100 . do
    return small .
  end
  return big .
end
```

Output

```
$ lowentc --check sizes.low
== check: ok ==
```

The `sizes` module exports the enum `kind` and `classify`, which returns one. If the importing side qualifies the variants with the module name, this happens.

examples/ch21/mistake_enumcase.low

```
module mistake_enumcase .
rem run: describe 5
rem run: describe 500

use sizes from "sizes.low" .

fn describe input n u64 . output u64 . do
  let k sizes.kind be sizes.classify n .
  match k do
    rem ✘ a variant with the module name in front; in this edition it matches any value
    case sizes.small do
      return 1 .
    end
    case sizes.big do
      return 2 .
    end
  end
end
```

Output

```
$ lowentc --run describe mistake_enumcase.low 5
describe(5) = 1
$ lowentc --run describe mistake_enumcase.low 500
describe(500) = 2
```

`describe 500` is `big` and answers 2, and `describe 5` answers 1: a qualified name is read as a variant. Until 2026-09-16 it was not — lowering read `case sizes.small` not as a variant but as **a slot that matches any value**, so the first arm took every value and 500 also answered 1. The checker narrowed qualified names to variants and lowering did not; the two layers now read the same tree. Writing only the variant name is still shorter, and the type of `k` decides which enum it belongs to.

examples/ch21/enumcase_fixed.low

```
module enumcase_fixed .
rem run: describe 5
rem run: describe 500

use sizes from "sizes.low" .

fn describe input n u64 . output u64 . do
  let k sizes.kind be sizes.classify n .
  match k do
    rem write only the variant name; the type (sizes.kind) decides which variant it is
    case small do
      return 1 .
    end
    case big do
      return 2 .
    end
  end
end
```

Output

```
$ lowentc --run describe enumcase_fixed.low 5
describe(5) = 1
$ lowentc --run describe enumcase_fixed.low 500
describe(500) = 2
```

A common misconception. Two modules must not import each other

examples/ch21/cycle_a.low

```
module cycle_a .
rem run: fa 3
rem run: fa 4

rem imports `cycle_b`, and `cycle_b` imports this module too
use cycle_b from "cycle_b.low" .

export fn fa input n u32 . output u32 .
  requires le n 10 .
do
  if eq n 0 . do return 0 . end
  return cycle_b.fb (sub n 1) .
end
```

Output

```
$ lowentc --run fa cycle_a.low 3
fa(3) = 1
$ lowentc --run fa cycle_a.low 4
fa(4) = 0
```

`cycle_a` imports `cycle_b` and `cycle_b` imports `cycle_a`, yet it translates and runs. Unlike C headers, where only what was read first is known, names are resolved after the whole translation unit is gathered — the same principle that makes the order of top-level declarations irrelevant. Still, modules that import each other are easier to read merged into one, or with the shared part moved into a third module. The recursion depth is bounded by the contract (`requires le n 10 .`).

21.8 This chapter's syntax at a glance

Shape	Meaning	Why
`module geom .`	this file is module `geom` (first line)	the module name is separate from the file name — `use` looks for it
`export fn manhattan … ` · `export struct point …`	make it visible outside	hidden by default — what is visible is a promise
`use geom from "geom.low" .`	import from a place relative to the declaring file	no search path — dependencies are in the source

`use allocs .`	import from the same unit or the standard library	otherwise `W-USE-EXTERNAL`
`geom.point` · `geom.manhattan a b`	qualify imported names with the module name	no spilling
`geom.helper` **(a hidden name)**	rejected (`E-VISIBILITY`)	qualifying does not open the door
one name declared twice · two imports with one name	rejected (`E-NAME-DUP` · `E-NAME-COLLISION`)	never decide quietly which one is reached
order of top-level declarations	irrelevant — they may call each other	a file is scanned; a body is read top to bottom
`use geom from "geom.low" as g .`	call the imported module `g` in this file	untangles long or clashing names — the module's real name stays
`export let …` · `export test …`	refused (`E-TOPLEVEL`)	export a constant through a `fn` that returns it
`case small` **on another module's enum**	write only the variant name	`case sizes.small` matches any value in this edition

Table 21.2 — Module syntax — shape · meaning · why it looks this way

Recap

One file is one module, and only what is `export`ed is visible outside. `use <module> from "<place>" .` imports from a place relative to the declaring file; without a place it looks in the same compilation unit or the standard library. There is no search path. Qualification does not reach hidden names, and declaring the same name twice is rejected. Top-level order does not matter, but locals in a body are declared first.

22 Generics — parameters fixed at translation time

In this chapter

22.1 Taking values and types at translation time 274
22.2 An instance per combination 276
22.3 Putting conditions on types 276
22.4 The type carries it, not a value 278
22.5 Common mistakes 279
22.6 This chapter's syntax at a glance 283

What to know first

chapter 8, Expressions · `size_of` and `comptime` are computed at translation time
chapter 20, Allocators and fixed memory · allocators are taken as `input comptime a type .` and swapped

Looking back

chapter 20's `two_from` accepted both `bump_bytes` and `bump_aligned`. Why was swapping allocators said to cost nothing?

> **A.** Because the allocator's **type** is fixed to a concrete type at translation, and code specific to that type is made for each call. There is no virtual function table and no indirect call. This chapter covers "parameters fixed at translation time" — `comptime` — in general.

The need for this chapter, and its context

Copying the same algorithm for every type multiplies the places to fix by the number of types. Passing comparison as a function pointer looks free at the call site but hides the cost of an indirect call. Lowent takes types and constants as **translation-time parameters** and makes a real instance for each combination used, and what gets made is visible at the call site. Traits (chapter 23) and `pipe` (chapter 24) both stand on this, so it is set up first as the tool of abstraction.

By the end of this chapter

You will learn to take values at translation time, as in `input comptime n u8 .`, and types, as in `input comptime t type .`. You will see that giving a value unknown at translation time is rejected, that a real instance is made for each combination used (monomorphisation), and that their number is the amount of code. You will also see how to put a condition on a type parameter with `requires <trait> t .`, and what happens when a type that does not satisfy it is given.

The questions this chapter answers

1 Why not infer type arguments? Couldn't `max_of a b` be worked out from the type of `a`?

22.1 Taking values and types at translation time

A parameter marked `comptime` must have its value fixed at translation time. Types can be taken this way too.

examples/ch22/sizes.low

```
module sizes .
rem run: report

fn bytes_for input comptime t type . input items u64 . output u64 .
  requires le items 1000000 .
do
  return mul (size_of t) items .
end

fn add_const input comptime n u8 . input a u8 . output u8 .
  requires le a 200 .
do
  return add a n .
end
```

```
fn report output u64 .
do
  let a u64 be bytes_for u8 100 .
  let b u64 be bytes_for u64 100 .
  let c u8 be add_const 7 10 .
  return add (add a b) (widen u64 c) .
end
```

Output

```
$ lowentc --run report sizes.low
report() = 917
```

- `bytes_for`'s `input comptime t type .` takes a type. `size_of t` in the body gives the size of that type at translation time. `bytes_for u8 100` is 100 and `bytes_for u64 100` is 800.
- `add_const`'s `input comptime n u8 .` takes a value. `add_const 7 10` is 17.
- Call sites write the type or constant **in front, like an ordinary argument**. There are no angle brackets (`<T>`) and no inference.

comptime parameters come at the very front of the head (even before capability inputs) (chapter 3), because the following input and output types use their names.

Giving a value unknown at translation time is rejected.

examples/ch22/rt_arg.low

```
module rt_arg .
rem expect: E-COMPTIME-ARG

fn add_const input comptime n u8 . input a u8 . output u8 .
  requires le a 200 .
do
  return add a n .
end

fn caller input k u8 . output u8 .
do
  return add_const k 4 .
end
```

Output

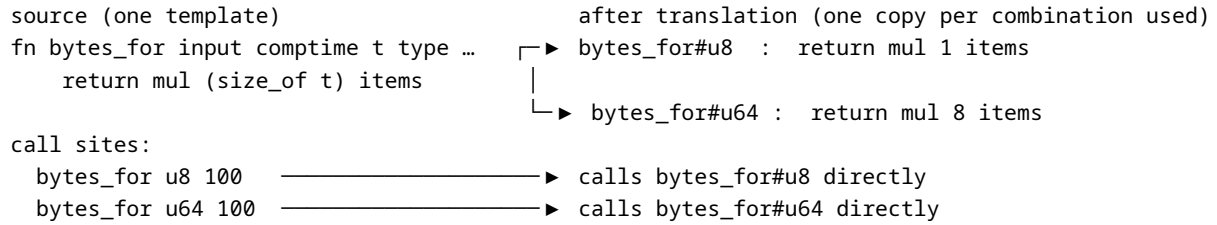
```
$ lowentc --check rt_arg.low
rt_arg.low:12:0 E-COMPTIME-ARG: this argument sits in a `comptime` parameter, so it must
 be a COMPILE-TIME CONSTANT (an integer literal, or a module `let`). A runtime value the
re makes the word `comptime` a lie — and the analysis, the folding and the monomorphisat
ion all believe it
```

`k` is only known at run time. Once the position says `comptime`, receiving a run-time value would make that word a lie. What may stand in a comptime position is an integer literal or a module-level `let` constant.

> **Q. Why not infer type arguments? Couldn't `max_of a b` be worked out from the type of `a`?**
>
> **A.** It could. But with inference, **what gets made** is not visible at the call site. The same line would call different instances depending on argument types, and you would have to follow the source to count them. Lowent chose to leave countable at the call site the cost that the number of combinations used is the amount of code made.

22.2 An instance per combination

The processor makes an op specific to each combination used. This is called **monomorphisation**. `bytes_for u8` and `bytes_for u64` are two different functions in native code. Because they are made after the type is fixed, sizes and operations are baked in as constants, and calls are direct.



The template's `size_of t` becomes the constants 1 and 8 in the copies. No "what is t?" question is left for run time.

The price is the amount of code. Calling with ten types makes ten copies. That cost does not hide; the types are written at the call sites, so the number of copies can be counted from the source.

22.3 Putting conditions on types

An op that takes a type usually needs that type to know how to do something. To pick a maximum, it must know how to compare. That condition is put as `requires <trait> <type> .`.

examples/ch22/bound.low

```
module bound .
rem run: bigger 3 9
rem run: bigger 12 5

trait ordered do
  less input a self . input b self . output bool .
end

struct score do
  satisfies ordered .
  v u64 .
end

fn score.less input a score . input b score . output bool .
do
  return lt (field a v) (field b v) .
```

```
end

fn max_of input comptime t type . input a t . input b t . output t .
  requires ordered t .
do
  if method a less b . do
    return b .
  end
  return a .
end

fn bigger input x u64 . input y u64 . output u64 .
do
  let m score be max_of score (make score do v x . end) (make score do v y . end) .
  return field m v .
end
```

Output

```
$ lowentc --run bigger bound.low 3 9
bigger(3, 9) = 9
$ lowentc --run bigger bound.low 12 5
bigger(12, 5) = 12
```

- `trait ordered` is the promise “has an op called `less`”. `score` declares it will satisfy that promise with `satisfies ordered .` and actually does so with `score.less` (chapter 23).
- `max_of`’s `requires ordered t .` is the type condition. The body trusts it and calls `method a less b`.
- `bigger` calls `max_of score` …. The processor makes a `score`-specific instance of `max_of`; in the emitted C you can see `max_of_score` in that function’s name.

Calling with a type that does not satisfy the condition is rejected.

examples/ch22/unsat.low

```
module unsat .
rem expect: E-BOUND-UNSAT

trait ordered do
  less input a self . input b self . output bool .
end

struct plain do
  v u64 .
end

fn max_of input comptime t type . input a t . input b t . output t .
  requires ordered t .
do
  if method a less b . do
    return b .
  end
  return a .
end

fn caller output u64 .
do
```

```
  let m plain be max_of plain (make plain do v 1 . end) (make plain do v 2 . end) .
  return field m v .
end
```

Output

```
$ lowentc --check unsat.low
13:0 E-BOUND-UNSAT: this generic was instantiated with a type that does NOT satisfy the
required trait. A bound is a PROMISE the callee relies on — instantiating past it would
make the callee's contract a lie (RFC-0021 §6.3). Add `satisfies <trait> .` to the struc
t, and the required ops as `fn <Type>.<name>`
```

A type condition is a promise, and letting a caller make an instance past it would make the callee's contract a lie. The diagnostic says to add `satisfies` and the required op to that type.

> **A common misconception. Generic code is slow**
>
> It depends on the language. Erasing types, boxing every value and dispatching through indirect calls is slow. Monomorphising becomes the same machine code as copying the code by hand for each type. Lowent monomorphises. The price is not speed but the amount of code, and that amount can be counted at the call sites.

22.4 The type carries it, not a value

C's `qsort` takes the comparison function **as a value**. Lowent has no first-class functions, and even if it did, the indirect call through a function pointer would hide. Instead **the type carries the comparison**. The head of the standard library's `sortgen` has that shape.

```
export trait ordered do
  less input a self . input b self . output bool . effects none .
end

export proc sort_by input comptime t type . input s mut slice t .
  output void .
  effects none .
  requires ordered t .
```

To change the ordering, use a type with a different `less`. Wrapping in a one-field struct keeps the layout unchanged, so it costs nothing. Descending order or multiple keys are also a matter of writing `less` that way. Instead of mode arguments, the type carries the meaning (chapter 34).

> **In practice. The type decides even the allocation effects**
>
> Chapter 15's `via` is where generics reach effects. `vecgen.append` is written `effects state via a .`, so monomorphised on a bump over borrowed bytes only `state` appears in that instance's signature, and monomorphised on `heap_bytes`, which carves from the heap, `heap state` appears. The type argument decides not only the algorithm but the **effects**. That is how one source serves both machines without an operating system and servers.

22.5 Common mistakes

Counter-example. Leaving out the type argument and expecting it to be inferred

examples/ch22/mistake_notypearg.low

```
module mistake_notypearg .
rem expect: E-MONO-NOTYPE

trait ordered do
  less input a self . input b self . output bool .
end

struct score do
  satisfies ordered .
  v u64 .
end

fn score.less input a score . input b score . output bool .
do
  return lt (field a v) (field b v) .
end

fn max_of input comptime t type . input a t . input b t . output t .
  requires ordered t .
do
  if method a less b . do
    return b .
  end
  return a .
end

fn bigger input x u64 . input y u64 . output u64 .
do
  rem ✘ the leading type argument `score` is missing --- nothing is inferred
  let m score be max_of (make score do v x . end) (make score do v y . end) .
  return field m v .
end
```

Output

```
$ lowentc --check mistake_notypearg.low
mistake_notypearg.low:30:0 E-MONO-NOTYPE: this op takes a TYPE as its first input (`inpu
t comptime t type .`) and the call does not give one. Nothing is inferred from the argum
ent types here: what is being built has to be visible at the call — write the type first
, as in `max_of score a b`. (Without it the instance is never built, which is why the to
ol used to say the op did not exist)
```

The first input of `max_of` is a type. It may look as if `score` could be worked out from the arguments, but Lowent does not infer — what gets built must be visible at the call. Leaving the leading position empty is `E-MONO-NOTYPE`, which asks for the type up front. The same spot used to say “there is no name `max_of`” (`E-IR-UNDEF`) — the op clearly exists, so that message misled.

Counter-example. Not stating, as a type condition, the behaviour the body uses

examples/ch22/mistake_nobound.low

```
module mistake_nobound .
rem expect: E-METHOD-UNDEF

trait ordered do
  less input a self . input b self . output bool .
end

struct plain do
  v u64 .
end

rem ✘ the body calls `less`, but the head has no `requires ordered t .`
fn max_of input comptime t type . input a t . input b t . output t .
do
  if method a less b . do
    return b .
  end
  return a .
end

fn caller output u64 .
do
  let m plain be max_of plain (make plain do v 1 . end) (make plain do v 2 . end) .
  return field m v .
end
```

Output

```
$ lowentc --check mistake_nobound.low
mistake_nobound.low:15:0 E-METHOD-UNDEF: no op of that name is associated with the recei
ver's type — declare it as `fn <type>.<name> input <recv> <type> . …` (RFC-0062)
23:0 N-MONO-SITE: …and THIS is the line that asked for that instance — the error above i
s inside a GENERIC template you did not write. The template's line is right; what was mi
ssing is WHERE it was instantiated. Fix the type argument here, or state the requirement
 on the template's boundary
```

The body of `max_of` calls `less`, but the head has no `requires ordered t .`. The mistake of calling it with `plain` then appears as `E-METHOD-UNDEF` on a line **inside the template**, and the tool adds `N-MONO-SITE` to say "this is the line that asked for the instance". `unsat.low`, which states the condition, reports the same mistake directly at the call with `E-BOUND-UNSAT`. A type condition is both a promise to callers and the mark that brings the diagnostic back to the right place.

Counter-example. Passing a size where a type belongs

examples/ch22/mistake_valuetype.low

```
module mistake_valuetype .
rem expect: E-IR-UNDEF

fn bytes_for input comptime t type . input items u64 . output u64 .
  requires le items 1000000 .
```

```
do
  return mul (size_of t) items .
end

fn total output u64 .
do
  rem ✘ a size (8) is passed where a type belongs --- write `u64`
  return bytes_for 8 100 .
end
```

Output

```
$ lowentc --check mistake_valuetype.low
mistake_valuetype.low:7:0 E-IR-UNDEF: size_of needs a SIZED type (a scalar, or a VIEWABL
E struct the unit declares) — it does not answer for slices or views, whose size is not
a property of the type
13:0 N-MONO-SITE: …and THIS is the line that asked for that instance — the error above i
s inside a GENERIC template you did not write. The template's line is right; what was mi
ssing is WHERE it was instantiated. Fix the type argument here, or state the requirement
 on the template's boundary
```

`u64` is 8 bytes, so passing 8 may seem fine, but `input comptime t type .` takes a **type**. This edition's tool tries to read the number 8 as a type and reports `E-IR-UNDEF` at `size_of t` inside the template. Write the type name, as in `bytes_for u64 100`. Needing the size is the template's business; the caller says what it is the size of.

Counter-example. Placing a `comptime` input after a data input

examples/ch22/mistake_comptimeorder.low

```
module mistake_comptimeorder .
rem expect: E-CLAUSE-ORDER

rem ✘ the `comptime` input comes after a data input
fn add_const input a u8 . input comptime n u8 . output u8 .
  requires le a 200 .
do
  return add a n .
end
```

Output

```
$ lowentc --check mistake_comptimeorder.low
mistake_comptimeorder.low:5:27 E-CLAUSE-ORDER: a `comptime` input comes after a data inp
ut. An op header has ONE order: `satisfies`/`lowdoc` · `vector`/`priority` · `comptime`
inputs · capability/region inputs · `using` · data inputs · `output` · `effects` · `link
`/`variadic` · `asm` · `access`/`inplace`/`invalidates`/`parallel`/`reduce` · `requires`
 · `ensures` · `errors` · `tests` (`--fmt` moves the non-input clauses for you; inputs a
re call positions, so reorder those and their call sites yourself)
```

A head has exactly one order. `comptime` inputs come first, ahead even of capability inputs, because the types of the following inputs and output must be able to use their names, and at

the call “what to build” should be read first. The `E-CLAUSE-ORDER` diagnostic shows the whole head order on one line.

A common misconception. A local `let` whose value never changes is a compile-time constant

examples/ch22/mistake_localconst.low

```
module mistake_localconst .
rem expect: E-COMPTIME-ARG

fn add_const input comptime n u8 . input a u8 . output u8 .
  requires le a 200 .
do
  return add a n .
end

fn seventeen output u8 .
do
  rem ✘ the value is always 7, but a `let` inside an op is a name that exists at run tim
e
  let k u8 be 7 .
  return add_const k 10 .
end
```

Output

```
$ lowentc --check mistake_localconst.low
mistake_localconst.low:14:0 E-COMPTIME-ARG: this argument sits in a `comptime` parameter
, so it must be a COMPILE-TIME CONSTANT (an integer literal, or a module `let`). A runti
me value there makes the word `comptime` a lie — and the analysis, the folding and the m
onomorphisation all believe it
```

`k` is always 7, but a `let` inside an op is a name that comes into being when the op is called. A `comptime` position accepts only integer literals and **module-level** `let`s. If the tool started following expressions to decide whether something is “a constant after all”, which expressions resolve at translation time would depend on how clever the tool is. Give a value you want as a constant a name at module level.

examples/ch22/module_const.low

```
module module_const .
rem run: seventeen

rem a module-level `let` is a compile-time constant
let step u8 be 7 .

fn add_const input comptime n u8 . input a u8 . output u8 .
  requires le a 200 .
do
  return add a n .
end

fn seventeen output u8 .
```

```
do
  return add_const step 10 .
end
```

Output

```
$ lowentc --run seventeen module_const.low
seventeen() = 17
```

22.6 This chapter's syntax at a glance

Shape	Meaning	Why
`input comptime t type .`	receive a type at translation time (first in the head)	later types use its name
`input comptime n u8 .`	receive a value at translation time	folded as a constant; checks disappear
`bytes_for u64 100·add_const 7 10`	write types and constants as leading arguments	no angle brackets, no inference — what is built is visible
`let step u8 be 7 .` (module level)	a named constant allowed in a `comptime` position	a `let` inside an op is a run-time name
`size_of t`	the size of a type at translation time	the size is fixed as a constant
`requires ordered t .`	type condition — the type must adopt the trait	otherwise `E-BOUND-UNSAT` at the call
`method a less b`	call the op the condition promises	monomorphised into a direct call
one concrete copy per combination used	monomorphisation	the cost is code size, not speed — counted in the source

Table 22.1 — Generic syntax — shape · meaning · why it looks this way

Recap

`comptime` parameters have their values fixed at translation time, and types are taken as `input comptime t type .`. Call sites write types and constants as leading arguments, with no inference. A run-time value in a comptime position is rejected. An instance is made for each combination used, calls are direct, and their number is the amount of code. `requires <trait> t .` is a type condition, and types that do not satisfy it are rejected. Behaviour such as comparison is carried by the type, not by a value.

23 Traits — one promise kept by many types

In this chapter

23.1 What they are for 285
23.2 Ops attached to types, and `method` 286
23.3 Traits with several ops 287
23.4 Signatures do not say `fn` or `proc` 288
23.5 When a trait is not satisfied 290
23.6 `via self` — more allocation effects only 291
23.7 Common mistakes 293
23.8 This chapter's syntax at a glance 296

What to know first

chapter 10, Aggregates · `struct` and `field`
chapter 15, Effects · the effects line narrows and spreads to callers
chapter 22, Generics · `requires <trait> t .` is a type condition

Looking back

What did `max_of` in chapter 22 call in its body, trusting `requires ordered t .`? And what happened when `plain`, which did not satisfy the condition, was given?

> **A.** It called `method a less b`. Giving `plain` was rejected with `E-BOUND-UNSAT`, because the condition is a promise and making an instance past it would make the callee's contract a lie. This chapter covers declaring and satisfying that promise — the trait — in full.

The need for this chapter, and its context

A rectangle and a square compute their areas differently, but the promise "can tell its area" is the same. Naming that promise lets you write, **once**, an op that accepts "anything that can tell its area". This language has no inheritance, so traits are the way to treat several types under one name. This chapter properly unfolds the tool already used for allocators (chapter 20) and sorting (chapter 22).

By the end of this chapter

You will learn to attach ops to types (`fn rect.area`) and call them with `method`. You will pick up how to declare a trait and satisfy it with `satisfies`, and how to use traits with several ops. You will see the rule that trait signatures do not say `fn` or `proc` and that the `effects` line decides the implementer, the diagnostics for failing to satisfy a trait, `via self`, and why a trait is not inheritance.

The questions this chapter answers

1 Can a trait op have a default implementation?

23.1 What they are for

examples/ch23/why.low

```
module why .
rem run: demo

trait shape do
  area input s self . output u64 .
end

struct rect do
  satisfies shape .
  w u64 .
  h u64 .
end

struct square do
  satisfies shape .
  side u64 .
end

fn rect.area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end

fn square.area input s square . output u64 .
do
  return mul (field s side) (field s side) .
end

fn double_area input comptime t type . input s t . output u64 .
  requires shape t .
do
  return mul 2 (method s area) .
end

fn demo output u64 .
do
  let r rect be make rect do w 2 . h 3 . end .
  let q square be make square do side 4 . end .
  return add (double_area rect r) (double_area square q) .
end
```

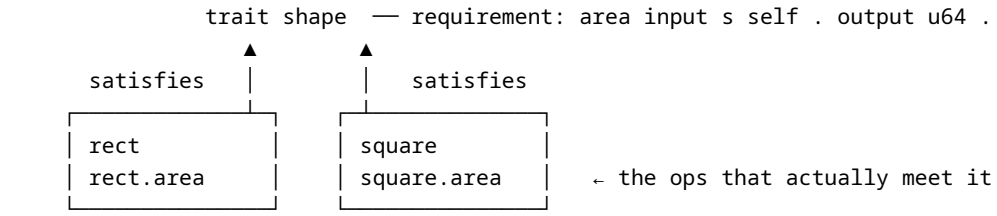
Output

```
$ lowentc --run demo why.low
demo() = 44
```

- `trait shape do … end` is the promise “has `area`”. `self` in a signature is the type that will satisfy the promise.
- `rect` and `square` write `satisfies shape .` in their bodies, declaring they will satisfy it.
- `fn rect.area` and `fn square.area` actually satisfy it. The `rect.` in the name means the op is attached to `rect`.

- `double_area` accepts **any shape**. `requires shape t .` pins it down to "only types that can tell their area".
- `method s area` calls the `area` attached to the type of `s` — `rect.area` for a `rect`, `square.area` for a `square`.

Think of a trait as a **qualification**. There is a requirement, "can tell its area", and only types that meet it may enter an op that asks for it.




```
double_area has requires shape t .  ── only types that meet shape
  double_area rect r    →  method s area  =  rect.area
  double_area square q  →  method s area  =  square.area
```

`double_area rect r` and `double_area square q` are each monomorphised to their own instance (chapter 22), so `method` is resolved at translation, not looked up at run time. There is no virtual function table.

23.2 Ops attached to types, and `method`

An op can be attached to a type even without a trait. Put the type name and a dot in front of the op name, and its first input is a value of that type. `method <value> <name> <args…>` calls that op. The receiver may be the result of another form — `method (method r grow 1) area`. If no such op is attached, the call is rejected.

examples/ch23/method_undef.low

```
module method_undef .
rem expect: E-METHOD-UNDEF

struct point do
  x u8 .
end

fn f input p point . output u8 .
do
  return method p nosuch .
end
```

Output

```
$ lowentc --check method_undef.low
method_undef.low:10:0 E-METHOD-UNDEF: no op of that name is associated with the receiver
's type — declare it as `fn <type>.<name> input <recv> <type> . …` (RFC-0062)
```

Calling a name that does not exist does not search upwards. Attached ops are a device for dividing the name space, not inheritance.

23.3 Traits with several ops

examples/ch23/many.low

```
module many .
rem run: demo 3 4
rem trap: demo_empty

trait shape do
  area input s self . output u64 .
  perimeter input s self . output u64 .
  grow input s self . input k u64 . output self .
  checked_area input s self . output u64 . effects panic .
end

struct rect do
  satisfies shape .
  w u64 .
  h u64 .
end

fn rect.area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end

fn rect.perimeter input s rect . output u64 .
do
  return mul 2 (add (field s w) (field s h)) .
end

fn rect.grow input s rect . input k u64 . output rect .
do
  return make rect do w (add (field s w) k) . h (add (field s h) k) . end .
end

proc rect.checked_area input s rect . output u64 . effects panic .
do
  if eq (field s w) 0 . do
    panic "empty rect" .
  end
  return mul (field s w) (field s h) .
end

proc demo input w u64 . input h u64 . output u64 . effects panic .
  requires le w 1000 .
  requires le h 1000 .
do
  let r rect be make rect do w w . h h . end .
  let big rect be method r grow 1 .
  return add (method big perimeter) (method big checked_area) .
end

proc demo_empty output u64 . effects panic .
do
  let r rect be make rect do w 0 . h 5 . end .
```

```
  return method r checked_area .
end
```

Output

```
$ lowentc --run demo many.low 3 4
demo(3, 4) = 38
$ lowentc --run demo_empty many.low
== ir diagnostics (1) ==
0:0 E-VM-PANIC: the program called `panic` — this is an unrecoverable stop, and it is NO
T a contract violation (the code chose to stop, it did not break a promise)
```

There are three things to know about writing them.

- One line per signature. A line starts with the op's **name** and then writes clauses in the same order as an op head — inputs, output, effects. The next name starts the next signature. Write as many as you like.
- When the output is `self`, as in `grow`, the implementer returns its own type (`rect`).
- An implementing op is named `<type>.<name>`.

`demo_empty` calls `checked_area` on a rectangle of width 0 and stops. The signature promised `effects panic`, so callers know it may stop.

> **In practice. A defect: effects of ops called through `method` do not spread**
>
> Running `--check` on the example above attaches `W-EFFECT-OVER` (`panic` declared but never performed) to `demo`. It is a false warning — `demo_empty` really does stop. While writing this book it turned out that the processor in this edition does not spread the effects of ops called through `method` to the caller. So even a pure `fn` can call an op that `panic`s through `method` and pass translation. Calling the same op directly as `rect.checked_area r` is correctly rejected. The behaviour the specification (chapter 15) requires is the direct call's; the `method` side is a defect. Until it is fixed, do not call effectful attached ops through `method` inside pure ops.

23.4 Signatures do not say `fn` or `proc`

Writing `fn` or `proc` in a trait signature is rejected.

examples/ch23/sig_kind.low

```
module sig_kind .
rem expect: E-TRAIT-SIG

trait shape do
  fn area input s self . output u64 .
end
```

Output

```
$ lowentc --check sig_kind.low
5:3 E-TRAIT-SIG: a trait signature does not say `fn` or `proc` — write `area input s sel
f . output u64 .`. Its `effects` line says what the op may do (none = pure), and the imp
lementation chooses `fn` or `proc` (§6.11.2)
```

What the op may do is decided by the signature's `effects` line.

Signature's `effects`	Implementer
none	a `fn`, or a `proc` that writes `effects`
present, e.g. `effects panic`	a `proc` declaring those effects or fewer; a `fn` if it uses none

Table 23.1 — A signature's `effects` and the implementer

A `proc` **without** an `effects` line is not narrowed and reads as able to do anything (chapter 5). So implementing an effect-free signature with such a `proc` is rejected.

examples/ch23/proc_noeff.low

```
module proc_noeff .
rem expect: E-TRAIT-EFFECT

trait shape do
  area input s self . output u64 .
end

struct rect do
  satisfies shape .
  w u64 .
  h u64 .
end

proc rect.area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end
```

Output

```
$ lowentc --check proc_noeff.low
9:0 E-TRAIT-EFFECT: the implementation is a `proc` with NO `effects` clause, so it may d
o anything — more than the trait declares. Write the same `effects` line the trait has (
or fewer), or make it a `fn` if it is pure
```

Declaring more effects than the signature is rejected too.

examples/ch23/more_effect.low

```
module more_effect .
rem expect: E-TRAIT-EFFECT

trait shape do
  checked_area input s self . output u64 . effects panic .
end

struct rect do
  satisfies shape .
  w u64 .
  h u64 .
end

proc rect.checked_area input s rect . output u64 . effects panic wait .
do
```

```
  if eq (field s w) 0 . do
    panic "empty rect" .
  end
  return mul (field s w) (field s h) .
end
```

Output

```
$ lowentc --check more_effect.low
9:0 E-TRAIT-EFFECT: the implementation has an EFFECT the trait does not declare. A calle
r reasons against the TRAIT's contract — if the implementation does more, that reasoning
 is a lie (this is exactly what makes dynamic dispatch safe or unsafe)
more_effect.low:15:1 W-EFFECT-OVER: this op DECLARES `wait` but never PERFORMS it. A dec
lared effect is a cost the caller must budget for (a pure caller cannot call an `io` op)
. Drop it, or — if a future version will perform it — say so (RFC-0007)
```

Callers reason against **the trait's contract**. If the implementer does something outside that contract, the reasoning goes wrong. A trait whose names match but whose effects differ cannot carry a contract.

23.5 When a trait is not satisfied

Diagnostic	Meaning
E-TRAIT-UNDEF	`satisfies` names an undeclared trait
E-TRAIT-MISSING	one op in the list is missing
E-TRAIT-SIG	`fn`/`proc` written in a signature, or the op exists with a different parameter count
E-TRAIT-EFFECT	the implementing op has more effects than the signature
E-TRAIT-RECV	`satisfies` was written on an op instead of a type

Table 23.2 — Where a trait is not satisfied

examples/ch23/missing.low

```
module missing .
rem expect: E-TRAIT-MISSING

trait shape do
  area input s self . output u64 .
  perimeter input s self . output u64 .
end

struct rect do
  satisfies shape .
  w u64 .
  h u64 .
end

fn rect.area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end
```

Output

```
$ lowentc --check missing.low
10:0 E-TRAIT-MISSING: this type claims to satisfy a trait, but the trait REQUIRES an op
that the type does not have. Declare it as `fn <Type>.<name> …` (RFC-0062). A claim that
 is not checked is the defect this language exists to remove
```

`satisfies` is not a comment. The moment you write that you will satisfy a trait, the processor checks the whole list.

> **Q. Can a trait op have a default implementation?**
>
> **A.** No. A trait is only a list and holds no code. If several types want to share an implementation, write it as a generic op (like `double_area`) and have the types satisfy only the minimum that op needs. With default implementations you have to search upwards to find where a type's `area` came from, and that is the entropy inheritance creates.

23.6 `via self` — more allocation effects only

Some promises, like the allocator trait, have allocation effects that differ by implementation. When a signature's effects line says `via self`, the implementer may declare **more** allocation-family effects (`alloc`, `heap`, `lock`, `atomic`) than the signature. Declaring more of any other effect is still `E-TRAIT-EFFECT`.

```
export trait byte_allocator do
  reserve input s self . input n u64 . output option mut slice u8 . effects state via self .
  grow input s self . input old mut slice u8 . input newn u64 . output option mut slice u8 .
effects state via self .
  used input s self . output u64 . effects state .
end
```

A bump allocator's `reserve` is just `state`, while `heap_bytes`, carving from the heap, has `heap state`. That difference rises through a generic op's `via a` all the way to its callers (chapter 20).

The same device serves promises that are not allocators. Two actors satisfy a ticket-issuing trait: one only counts, the other carves bytes from the fixed window for every ticket.

examples/ch23/viaself.low

```
module viaself .
rem run: main

rem a promise to hand out tickets; an implementation may add allocation-family effects
trait ticketer do
  issue input s self . output u64 . effects state via self .
end

rem only counts, so its effect is just state
actor counter do
  satisfies ticketer .
  state do
    next u64 .
  end
  proc issue output u64 . effects state . do
```

```
    set next (add next 1) .
    return next .
  end
end

rem carves 8 bytes from the fixed window per ticket, so it adds alloc
actor carver do
  satisfies ticketer .
  state do
    root cap allocator .
    count u64 .
  end
  proc issue output u64 . effects alloc state . do
    let g option mut slice u8 . . be alloc_bytes root capacity 8 .
    guard is_some g . else return 0 .
    set count (add count 1) .
    return mul count 10 .
  end
end

rem takes any implementation; via t carries that implementation's effects up to here
proc issue_two
  input comptime t type .
  input who t .
  output u64 .
  effects state via t .
  requires ticketer t .
do
  let a u64 be send who issue .
  let b u64 be send who issue .
  return add a b .
end

proc main input al cap allocator . output u8 . effects alloc state .
do
  var c counter be spawn actor counter .
  var k carver be spawn actor carver .
  let x u64 be issue_two counter c .
  let y u64 be issue_two carver k .
  return narrow u8 (add x y) .
end
```

Output

```
$ lowentc --run main viaself.low
main() = 33
```

- The signature is `issue input s self . output u64 . effects state via self ..counter` satisfies it with `effects state .` and `carver` with `effects alloc state ..alloc` belongs to the allocation family, so `via self` allows it.
- `issue_two` says `effects state via t ..` The instance specialised to `counter` has the effect `state`; the one specialised to `carver` has `alloc state`. The effects are exact **per instance**.
- That is why `main` must receive `cap allocator` and declare `alloc`. Without the line that calls with `carver`, `main` would need no `alloc`.
- The answer 33 is `counter`'s 1 + 2 plus `carver`'s 10 + 20.

`carver`'s state has a **capability field**, `root cap allocator` .. The rule for that field is covered in chapter 25.

> **A common misconception. Satisfying a trait inherits something from it**
>
> Nothing is inherited. Satisfying a trait is only the **fact** that "these ops exist", and no hierarchy arises between types. `rect` and `square` have no relation to each other even after satisfying `shape`. This language has no inheritance.

23.7 Common mistakes

> **Counter-example. Writing the name before the value after `method`**

examples/ch23/mistake_methodorder.low

```
module mistake_methodorder .
rem expect: E-METHOD-UNDEF

struct rect do
  w u64 .
  h u64 .
end
fn rect.area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end
fn f output u64 .
do
  let r rect be make rect do w 2 . h 3 . end .
  rem ✘ after `method` comes the value first, then the name
  return method area r .
end
```

Output

```
$ lowentc --check mistake_methodorder.low
mistake_methodorder.low:16:0 E-METHOD-UNDEF: no op of that name is associated with the r
eceiver's type — declare it as `fn <type>.<name> input <recv> <type> . …` (RFC-0062)
```

Translating `r.area()` from an object-oriented language backwards easily gives `method area r`. `method` takes **the value first**, because the value's type decides which attached op to look for. With the name first, the tool reads `area` as the value, looks for an op `r` on its type, and reports `E-METHOD-UNDEF`. Write `method r area`.

> **Counter-example. Leaving the type prefix off the name of the implementing op**

examples/ch23/mistake_noprefix.low

```
module mistake_noprefix .
rem expect: E-TRAIT-MISSING

trait shape do
  area input s self . output u64 .
```

```
end
struct rect do
  satisfies shape .
  w u64 .
  h u64 .
end
rem ✘ no `rect.` in the name --- an ordinary op, not attached to the type
fn area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end
```

Output

```
$ lowentc --check mistake_noprefix.low
8:0 E-TRAIT-MISSING: this type claims to satisfy a trait, but the trait REQUIRES an op t
hat the type does not have. Declare it as `fn <Type>.<name> …` (RFC-0062). A claim that
is not checked is the defect this language exists to remove
```

`fn area input s rect .` is an **ordinary op** that takes a `rect`, not an op attached to `rect`. The trait looks for `rect.area`, so this is `E-TRAIT-MISSING`. The tool does not attach ops by looking at their input types because, when several ordinary ops take the same type, the name alone should tell you which one fulfils the promise. This diagnostic lacks the file name; its line number points at the type declaration that says `satisfies`.

Counter-example. Treating a type as adopting a trait just because the op exists

examples/ch23/mistake_nosatisfies.low

```
module mistake_nosatisfies .
rem expect: E-BOUND-UNSAT

trait shape do
  area input s self . output u64 .
end
rem ✘ `rect.area` exists, but `satisfies shape .` is not written
struct rect do
  w u64 .
  h u64 .
end
fn rect.area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end
fn double_area input comptime t type . input s t . output u64 .
  requires shape t .
do
  return mul 2 (method s area) .
end
fn f output u64 .
do
  let r rect be make rect do w 2 . h 3 . end .
  return double_area rect r .
end
```

Output

```
$ lowentc --check mistake_nosatisfies.low
17:0 E-BOUND-UNSAT: this generic was instantiated with a type that does NOT satisfy the
required trait. A bound is a PROMISE the callee relies on — instantiating past it would
make the callee's contract a lie (RFC-0021 §6.3). Add `satisfies <trait> .` to the struc
t, and the required ops as `fn <Type>.<name>`
```

With `rect.area` in place, `rect` is indeed "a type that reports an area". Without `satisfies shape .`, though, `double_area rect r` is `E-BOUND-UNSAT`. Under duck typing, where matching shape is enough, an op that happens to share a name is wrongly read as fulfilling the promise. `satisfies` declares "I will keep this contract", and only that declaration makes the processor check the whole list.

Counter-example. Putting the receiver somewhere other than the first input

examples/ch23/mistake_recvlast.low

```
module mistake_recvlast .
rem expect: E-METHOD-RECV

struct rect do
  w u64 .
  h u64 .
end
rem ✘ the receiver `s` is not the first input
fn rect.scaled input k u64 . input s rect . output u64 .
  requires le k 100 .
do
  return mul k (field s w) .
end
fn f output u64 .
do
  let r rect be make rect do w 2 . h 3 . end .
  return method r scaled 5 .
end
```

Output

```
$ lowentc --check mistake_recvlast.low
mistake_recvlast.low:9:0 E-METHOD-RECV: an op attached to a type takes that type as its
FIRST input — that is what `method <value> <name> …` passes. Here the receiver is not fi
rst, so a `method` call hands the value to the wrong parameter: it used to pass `--check
` and stop at run time with `E-VM-TYPE`. Move the receiver to the first `input` clause
```

The first input of an attached op is the receiver. `method r scaled 5` puts `r` in the first position and 5 in the second. `scaled` takes the first as `k` and the second as `s`, so it tries to multiply a struct and stops. Such a head is rejected at translation time with `E-METHOD-RECV`. Always put `input s rect .` first in an attached op.

A common misconception. An op attached to a type can only be called through `method`

examples/ch23/direct_call.low

```
module direct_call .
rem run: f

struct rect do
  w u64 .
  h u64 .
end
fn rect.area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end
fn f output u64 .
do
  let r rect be make rect do w 2 . h 3 . end .
  rem the same op called two ways --- directly by name, and through `method`
  return add (rect.area r) (method r area) .
end
```

Output

```
$ lowentc --run f direct_call.low
f() = 12
```

`rect.area r` calls it directly by name, and `method r area` finds the same op from the value's type. They are the same op, and the results are 6 and 6. `method` earns its place where the type is a generic parameter and its name cannot be written (`method s area` in `double_area`). Where the type is known, a direct call also propagates effects correctly (see the defect case above).

23.8 This chapter's syntax at a glance

Shape	Meaning	Why
`fn rect.area input s rect . …`	attach an op to a type — the first input is the receiver	a way to divide the name space, not inheritance
`method r area` · **`rect.area r`**	call by the value's type · call directly by name	fixed at translation time — no virtual table
`trait shape do area input s self . output u64 . end`	the list of ops a type must have	`self` is the adopting type itself
signature line: name · inputs · output · effects	same order as an op head; no `fn`/`proc`	the effects line caps the implementer
`struct rect do satisfies shape . … end`	declare that this type keeps the promise	the declaration triggers the full check
`requires shape t .`	type condition of a generic op	a type that does not adopt it: `E-BOUND-UNSAT`
`effects state via self .` (signature)	only allocation effects may be added	allocators differ in effects
`E-TRAIT-MISSING` · **`-SIG`** · **`-EFFECT`** · **`-RECV`** · **`-UNDEF`**	one diagnostic per way of falling short	`satisfies` is not a comment

`effects state via t` . **(generic op)**	inherits the extra effects the specialised type declared with `via self`	`state` for `counter`, `alloc state` for `carver`

Table 23.3 — Trait syntax — shape · meaning · why it looks this way

Recap

`fn <type>.<name>` attaches an op to a type and `method` calls it. A trait is a list of ops a type must have, and a type declares it satisfies one with `satisfies` in its body. Signatures start with a name and write inputs, output and effects in order, without `fn` or `proc`. A signature's `effects` is the upper bound on the implementer's effects. Mismatches are rejected with `E-TRAIT-*`, and `via self` allows only more allocation-family effects. A trait is not inheritance.

24 `pipe` — one line for each thing you mean to do

In this chapter

24.1 The same work, two shapes 298
24.2 Stages and terminals 299
24.3 Walking once is the definition 304
24.4 Reading only as much as needed 305
24.5 The built-in `map` · `filter` with the same names 305
24.6 Common mistakes 306
24.7 This chapter's syntax at a glance 310

What to know first

chapter 7, Flow · `for` walks slices and counting loops use `while`
chapter 9, Sequences · elements can be written only through a `mut slice`
chapter 22, Generics · named ops are passed instead of values

Looking back

When chapter 7 asked "how do you write a loop counting from 0 to `n` with `for`?", what did it say takes care of filtering, counting and collecting over a slice?

> **A.** `pipe`. Counting loops are written with `while` and `var`, and filtering, transforming and collecting over a slice with `pipe`. This chapter covers that `pipe`.

The need for this chapter, and its context

A loop walking a slice almost always has the same skeleton: keep an index, check for the end, take an element, test it, accumulate something, advance the index. Written by hand, **what you mean to do** ("keep only the digits and count them") gets buried among indices and counters, and off-by-one mistakes happen there. `pipe` hands the skeleton to the language and lets people write only what they mean, one word per line. And it still walks once, just like the hand-written loop. As the last chapter of Part VI on abstraction, it shows an abstraction that hides no cost.

By the end of this chapter

You will learn the shape `pipe <source> do <stages…> <terminal> end` and its seven stages and five terminals. You will pick up how to pass named ops to stages, how to collect into the caller's buffer with `collect into`, and that `fold`, `any`, `all` and `take` read only as much as needed. You will also understand that walking once without intermediate arrays is not an optimisation but **the definition** of `pipe`.

The questions this chapter answers

1. An op that uses `collect into` has `effects none` in its head. It changes the caller's buffer — is that not an effect?

24.1 The same work, two shapes

examples/ch24/digits.low

```
module digits .
rem run: digits_loop [97,49,50,98,51]
rem run: digits_pipe [97,49,50,98,51]

fn is_digit input c u8 . output bool .
do
  return and (ge c 48) (le c 57) .
end

fn digits_loop input s slice u8 . output u64 .
do
  var n u64 be 0 .
  var i u64 be 0 .
  while lt i (len s) . do
    if is_digit (index s i) . do
      set n (add n 1) .
    end
    set i (add i 1) .
  end
  return n .
end

fn digits_pipe input s slice u8 . output u64 .
do
```

```
  return pipe s do
    filter is_digit .
    count .
  end .
end
```

Output

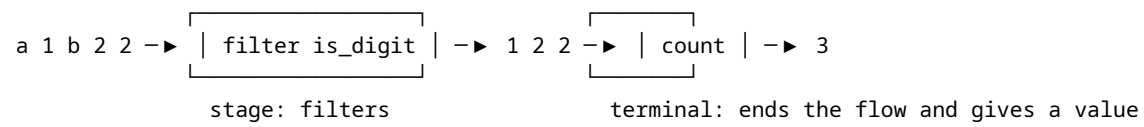
```
$ lowentc --run digits_loop digits.low [97,49,50,98,51]
digits_loop([97,49,50,98,51]) = 3
  arg0 (written) = [97,49,50,98,51]
$ lowentc --run digits_pipe digits.low [97,49,50,98,51]
digits_pipe([97,49,50,98,51]) = 3
  arg0 (written) = [97,49,50,98,51]
```

The two ops give the same answer. `digits_loop` has a person line up the counter `n`, the index `i`, the end condition and the increments. `digits_pipe` is two lines.

- `filter is_digit .` — keeps only elements for which `is_digit` is true.
- `count .` — counts what is left. The word that ends the flow (the **terminal**) is exactly one.

`pipe` is a statement, but when it ends with a value-producing terminal (`count`, `fold`, `any`, `all`) it can be used like an expression, as in `return pipe s do … end .` — the final stop belongs to the `return` statement (`end` closes only the `pipe` block).

Drawn as a tube that elements flow through:



Elements pass through the tube one at a time. No separate “array of just the digits” is built in between.

24.2 Stages and terminals

Word	Shape of the op	What it does	End · empty flow
`filter <op>`	`t → bool`	Passes only elements for which the op is true	—
`map <op>`	`t → u`	Passes the op’s answer for each element; the type may become `u`	—
`take <n>`	—	Passes the first `n`	**stops walking** after `n`
`skip <n>`	—	Drops the first `n`	—
`enumerate <op>`	`(u64, t) → u`	Gives the position (from 0) and the element to the op	—
`zip <slice> <op>`	`(t, t2) → u`	Gives the element and the other slice’s element	ends when the shorter side ends

		at the same place to the op	
`scan <init> <op>`	`(a, t) → a`	Passes the accumulator **after** each step (the initial value is not passed)	—
`collect into <buffer>`	—	**Terminal.** Fills the buffer from the front and gives **how many it stored** (`u64`)	a buffer too short is refused at translation when the lengths are known (`E-COLLECT-FULL`), otherwise the run stops
`fold <init> <op>`	`(a, t) → a`	**Terminal.** Gives the last accumulator (`a`)	empty: the initial value
`count`	—	**Terminal.** Gives the number of elements (`u64`)	empty: 0
`any <op>`	`t → bool`	**Terminal.** Is any element true	stops at the first true · empty: false
`all <op>`	`t → bool`	**Terminal.** Are all elements true	stops at the first false · empty: true

Table 24.1 — The words of `pipe` — element type `t`, accumulator type `a`

The op given to `filter`·`any`·`all` is a **predicate** and must answer `bool`. An op that answers a number is refused with `E-PIPE-PRED` — there is no rule here that reads a nonzero number as true (the same reason as `cast` in chapter 13).

Used as a value — `return pipe … end .` — a `pipe` is worth what its terminal gives. Each stage hands elements on to the next line, and the terminal takes them last and makes one value.

```
xs = [1, 5, 2, 7]
     │
     ▼ filter big      (big = gt x 2)      5, 7          1 and 2 are dropped here
     ▼ map dbl         (dbl = x + x)       10, 14
     ▼ collect into out                    out = [10, 14, …]   answer = 2 (how many stored)
```

examples/ch24/stages.low

```
module stages .
rem run: big_doubled [1,2,3,4,5] [0,0,0,0,0]
rem run: sum_big [1,2,3,4,5]
rem run: has_zero [4,0,9]
rem run: all_small [1,2,3]
rem run: middle [10,20,30,40] [0,0]

fn over2 input a u8 . output bool .
do
  return gt a 2 .
end

fn dbl input a u8 . output u8 .
do
  return wrap_add a a .
end
```

```
fn addu input acc u64 . input x u8 . output u64 .
do
  return wrap_add acc (widen u64 x) .
end

fn is_zero input a u8 . output bool .
do
  return eq a 0 .
end

fn under10 input a u8 . output bool .
do
  return lt a 10 .
end

proc big_doubled input xs slice u8 . input out mut slice u8 . output u64 . effects none
.
do
  pipe xs do
    filter over2 .
    map dbl .
    collect into out .
  end
  return len out .
end

fn sum_big input xs slice u8 . output u64 .
do
  return pipe xs do
    filter over2 .
    fold 0 addu .
  end .
end

fn has_zero input xs slice u8 . output bool .
do
  return pipe xs do
    any is_zero .
  end .
end

fn all_small input xs slice u8 . output bool .
do
  return pipe xs do
    all under10 .
  end .
end

proc middle input xs slice u8 . input out mut slice u8 . output u64 . effects none .
do
  pipe xs do
    skip 1 .
    take 2 .
    collect into out .
  end
```

```
  return len out .
end
```

Output

```
$ lowentc --run big_doubled stages.low [1,2,3,4,5] [0,0,0,0,0]
big_doubled([1,2,3,4,5], [6,8,10,0,0]) = 5
  arg0 (written) = [1,2,3,4,5]
  arg1 (written) = [6,8,10,0,0]
$ lowentc --run sum_big stages.low [1,2,3,4,5]
sum_big([1,2,3,4,5]) = 12
  arg0 (written) = [1,2,3,4,5]
$ lowentc --run has_zero stages.low [4,0,9]
has_zero([4,0,9]) = 1
  arg0 (written) = [4,0,9]
$ lowentc --run all_small stages.low [1,2,3]
all_small([1,2,3]) = 1
  arg0 (written) = [1,2,3]
$ lowentc --run middle stages.low [10,20,30,40] [0,0]
middle([10,20,30,40], [20,30]) = 2
  arg0 (written) = [10,20,30,40]
  arg1 (written) = [20,30]
```

- `big_doubled` keeps only elements greater than 2 (`filter`), doubles them (`map`) and puts them into `out` (`collect into`). In the result `[6,8,10,0,0]`, positions not filled stay as they were. The buffer is **given by the caller**; `pipe` does not allocate.
- `fold 0 addu` in `sum_big` starts from 0 and accumulates with `addu`.
- `has_zero` is `any is_zero`, and `all_small` is `all under10`.
- `middle` puts the middle two elements into `out` with `skip 1` and `take 2`.

A stage is given a **named op**. This language has no anonymous functions (lambdas). A stage op takes one element and receives no capabilities. The ops of `fold` and `scan` take the accumulator and an element.

Without tuples, how are positions or pairs handled? The pair is **not built but passed to the op as arguments**.

examples/ch24/pairs.low

```
module pairs .
rem run: with_index [10,10,10] [0,0,0]
rem run: added [1,2,3] [10,20,30] [0,0,0]
rem run: running [1,2,3,4] [0,0,0,0]

fn idxadd input i u64 . input e u8 . output u8 .
do
  return wrap_add e (narrow_wrap u8 i) .
end

fn addb input a u8 . input b u8 . output u8 .
do
  return wrap_add a b .
end

proc with_index input xs slice u8 . input out mut slice u8 . output u64 . effects none .
```

```
do
  pipe xs do
    enumerate idxadd .
    collect into out .
  end
  return 0 .
end

proc added input xs slice u8 . input ys slice u8 . input out mut slice u8 . output u64 .
 effects none .
do
  pipe xs do
    zip ys addb .
    collect into out .
  end
  return 0 .
end

proc running input xs slice u8 . input out mut slice u8 . output u64 . effects none .
do
  pipe xs do
    scan 0 addb .
    collect into out .
  end
  return 0 .
end
```

Output

```
$ lowentc --run with_index pairs.low [10,10,10] [0,0,0]
with_index([10,10,10], [10,11,12]) = 0
  arg0 (written) = [10,10,10]
  arg1 (written) = [10,11,12]
$ lowentc --run added pairs.low [1,2,3] [10,20,30] [0,0,0]
added([1,2,3], [10,20,30], [11,22,33]) = 0
  arg0 (written) = [1,2,3]
  arg1 (written) = [10,20,30]
  arg2 (written) = [11,22,33]
$ lowentc --run running pairs.low [1,2,3,4] [0,0,0,0]
running([1,2,3,4], [1,3,6,10]) = 0
  arg0 (written) = [1,2,3,4]
  arg1 (written) = [1,3,6,10]
```

`enumerate idxadd` gives `idxadd` the position and the element. `zip ys addb` gives `addb` an element of `xs` and the element of `ys` at the same position; when the shorter one ends the whole flow ends, so pairs always match. `scan 0 addb` passes the running sum on as the element. Since pairs are never built and immediately taken apart, there is no hidden allocation.

> **Q. An op that uses `collect into` has `effects none` in its head. It changes the caller's buffer — is that not an effect?**

> **A.** The specification's examples also write `effects none`. `collect into` does not **require** `effects state`, but if you write it the processor counts the write to caller storage, so no "declared but not performed" warning appears — the same judgement it makes for writes through a `mut` or

> `mut_ref` parameter. Either way, the caller knows writes may happen where it hands over a `mut slice` (chapter 12).

24.3 Walking once is the definition

A single `pipe` runs **in one pass**, and no intermediate arrays are created between stages. This is not an optimisation the processor may try but the definition of `pipe`, and a processor that creates intermediate arrays does not conform.

Why make it the definition? If fusion were an optimisation, "how far fusion goes" would differ between processors, and users would hit **cliffs** — change one line and suddenly an intermediate array appears and things slow down. That cliff is invisible in the source. Lowent **left operations that cannot be fused out of the stage words entirely**. That is why operations needing to see everything, such as sorting, are not stages.

examples/ch24/bad_stage.low

```
module bad_stage .
rem expect: E-PIPE-STAGE

fn is_digit input c u8 . output bool .
do
  return and (ge c 48) (le c 57) .
end

fn broken input s slice u8 . output u64 .
do
  return pipe s do
    sort .
    count .
  end .
end
```

Output

```
$ lowentc --check bad_stage.low
bad_stage.low:12:0 E-PIPE-STAGE: this word is not a `pipe` stage. The stage vocabulary i
s CLOSED (RFC-0010 G5/D-A): stages are `filter <op> .`, `map <op> .`, `take <n> .`, `ski
p <n> .`, `enumerate <op> .`, `zip <other> <op> .` and `scan <init> <op> .`, and the pip
eline ends with exactly one terminal — `collect into <mut slice> .`, `fold <init> <op> .
`, `count .`, `any <op> .` or `all <op> .`. Fusion here is the MEANING, not an optimizat
ion, so anything that could not fuse into the single loop is not writable as a stage — u
se an explicit loop, or break the pipeline with an explicit intermediate collect
```

The list of stages is closed. `pipe` being a statement rather than an expression has the same reason: as an expression, stages could be detached and passed as values, and where one stream ends would not be visible in the source. `do … end` makes that boundary visible.

24.4 Reading only as much as needed

`any` stops at the first element giving true, `all` at the first giving false, and `take n` after passing `n` elements. Elements after the stop are not read, and stages are not run for them. `has_zero [4,0,9]` never looks at 9. Only because this property is written into the meaning can `pipe` handle sources without an end.

A terminal ends the flow. A stage after the terminal is rejected.

examples/ch24/no_terminal.low

```
module no_terminal .
rem expect: E-PIPE-NO-TERMINAL

fn is_digit input c u8 . output bool .
do
  return and (ge c 48) (le c 57) .
end

fn broken input s slice u8 . output u64 .
do
  return pipe s do
    count .
    filter is_digit .
  end .
end
```

Output

```
$ lowentc --check no_terminal.low
no_terminal.low:13:0 E-PIPE-NO-TERMINAL: a `pipe` stage may not come AFTER the terminal
— the terminal ends the pipeline (RFC-0010 G1)
```

> **A common misconception. `pipe` is convenient but slower than a hand-written loop**
>
> In many languages iterator chains leave intermediate objects or indirect calls behind, or hope the optimiser removes them. For `pipe`, walking once is the definition, and stage ops are monomorphised direct calls. One `pipe` lowers to one loop, and its costs arise at the same places as in a hand-written loop. The words were chosen so that the convenient way and the fast way do not diverge.

24.5 The built-in `map` · `filter` with the same names

`map` and `filter` also exist outside `pipe`. These are not stages but **built-in operations that copy one slice into another in a single statement**, and they take three arguments — `map <sink> <op> <source> .` · `filter <sink> <op> <source> .`.

```
module sink_map .

fn dbl input a u8 . output u8 . do return wrap_add a a . end

proc doubled input xs slice u8 . input out mut slice u8 . output u64 . effects none .
do
```

```
  map out dbl xs .
  return 0 .
end
```

Putting `[1,2]` into a three-slot `out` gives `[2,4,0]`. The other way round, putting `[1,2,3]` into two slots stops the run as the third is written — a full sink never drops the rest in silence (the same as `collect into`). The sink must be a `mut slice` (`E-MAP-SINK`) and the element a scalar (`E-MAP-ELEM`). The word is the same, but inside `pipe` it takes one op — the number of arguments tells which is meant.

24.6 Common mistakes

Counter-example. Writing an expression in a stage to imitate a lambda

examples/ch24/mistake_lambda.low

```
module mistake_lambda .
rem expect: E-FOLD-OP

fn count_big input xs slice u8 . output u64 .
do
  return pipe xs do
    rem ✘ imitates an anonymous function --- a stage takes one named op
    filter gt 2 .
    count .
  end .
end
```

Output

```
$ lowentc --check mistake_lambda.low
mistake_lambda.low:8:0 E-FOLD-OP: this `pipe` stage names an op that does not exist
```

Carrying over `filter(x => x > 2)` from another language easily gives `filter gt 2`. A stage takes **one named op**, so this is `E-FOLD-OP` (the diagnostic only says “no such op”). Naming a condition may feel like extra work, but the name `over2` becomes the explanation of that line, and the same condition can be reused in other `pipe`s.

examples/ch24/lambda_fixed.low

```
module lambda_fixed .
rem run: count_big [1,2,3,4,5]

rem give the condition a name --- the name becomes its explanation
fn over2 input a u8 . output bool .
do
  return gt a 2 .
end

fn count_big input xs slice u8 . output u64 .
do
  return pipe xs do
    filter over2 .
    count .
```

```
  end .
end
```

Output

```
$ lowentc --run count_big lambda_fixed.low [1,2,3,4,5]
count_big([1,2,3,4,5]) = 3
  arg0 (written) = [1,2,3,4,5]
```

Counter-example. Collecting a `map` that produces wide values into a narrow buffer

examples/ch24/mistake_widecollect.low

```
module mistake_widecollect .
rem expect: E-TYPE-COLLECT

fn times1000 input a u8 . output u64 .
do
  return mul (widen u64 a) 1000 .
end

proc scale input xs slice u8 . input out mut slice u8 . output u64 . effects none .
do
  pipe xs do
    rem ✘ a `map` producing `u64` is collected into a `u8` buffer --- 1000 becomes 232
    map times1000 .
    collect into out .
  end
  return 0 .
end
```

Output

```
$ lowentc --check mistake_widecollect.low
mistake_widecollect.low:14:0 E-TYPE-COLLECT: this `collect into` would put a WIDER value
 into a narrower buffer, and that loses bits silently (measured: 1000 became 232). A con
version that loses value never happens implicitly here (§6.2.5) — put a `map` in front t
hat says which narrowing you mean (`narrow`, `narrow_wrap`, `narrow_sat`, `narrow_try`),
 or collect into a buffer of the produced width
```

`times1000` produces `u64`, but `out` is a `u8` buffer. A narrowing that loses value happens only where it is written (§6.2.5), so this is refused with `E-TYPE-COLLECT`. Until 2026-09-16 it passed and 1000 and 2000 were stored silently as 232 and 208. If narrowing is needed, write `narrow`, `narrow_wrap` or `narrow_sat` inside the op given to `map`, so the place where it may stop is visible.

Counter-example. Swapping the accumulator and the element in a `fold` op

examples/ch24/mistake_foldorder.low

```
module mistake_foldorder .
rem expect: E-FOLD-ORDER

rem ✘ takes the element first --- the accumulator arrives as `x`, the element as `acc`
```

```
fn add_small_swapped input x u8 . input acc u64 . output u64 .
  requires le acc 1000000 .
do
  guard lt x 100 . else return acc .
  return add acc (widen u64 x) .
end

fn sum_small_wrong input xs slice u8 . output u64 .
do
  return pipe xs do
    fold 0 add_small_swapped .
  end .
end
```

Output

```
$ lowentc --check mistake_foldorder.low
mistake_foldorder.low:15:0 E-FOLD-ORDER: the accumulator of a `fold`/`scan` is the FIRST
 input of its op and also its output, because each step computes `acc = op(acc, element)
`. Here the first input and the output are declared with different types, so the running
 total is handed to a parameter that cannot hold it — a `u8` first input takes a total o
f 600 as 88 and the answer is silently wrong. Write the op as `input acc <out-type> . in
put x <element-type> .`
```

`fold` gives its op **the accumulator first and the element second**. `add_small_swapped` takes them the other way round, so `x` receives the accumulator and `acc` receives the element. Each step is `acc = op(acc, element)`, so the first input must have the same type as the output; here they are `u8` and `u64`, and the program is refused with `E-FOLD-ORDER`. Always write the op of `fold` or `scan` in the order `input acc … . input x … .`.

Counter-example. Using a stage op that can stop, in a `fn`

examples/ch24/mistake_stageeffect.low

```
module mistake_stageeffect .
rem expect: E-EFFECT-CALC

proc nonzero input a u8 . output bool . effects panic .
do
  if eq a 0 . do panic "zero in input" . end
  return true .
end

rem ✘ the stage op can stop, yet the op using the `pipe` is written as a `fn`
fn count_checked input xs slice u8 . output u64 .
do
  return pipe xs do
    filter nonzero .
    count .
  end .
end
```

Output

```
$ lowentc --check mistake_stageeffect.low
mistake_stageeffect.low:12:1 E-EFFECT-CALC: this fn is declared pure but performs `panic
` — make it a `proc` with `effects …`, or remove the effect
```

A `pipe` is a loop that calls its stage ops, so the effects of a stage op spread to the op containing the `pipe`. `nonzero` may `panic`, so `count_checked` performs `panic` too, and as a `fn` it is `E-EFFECT-CALC`. Write it as `proc … effects panic .`, or instead of stopping, use a pure op that filters out the elements that do not meet the condition.

A common misconception. `collect into` drops the rest when the buffer is full

examples/ch24/short_buffer.low

```
module short_buffer .
rem run: big_doubled [1,3,4] [0,0]
rem trap: big_doubled [3,4,5,6] [0,0]

fn over2 input a u8 . output bool .
do
  return gt a 2 .
end

fn dbl input a u8 . output u8 .
do
  return wrap_add a a .
end

rem the buffer has two slots --- two remaining elements fit; with four, the run stops as
 the third is stored
proc big_doubled input xs slice u8 . input out mut slice u8 . output u64 . effects none
.
do
  pipe xs do
    filter over2 .
    map dbl .
    collect into out .
  end
  return len out .
end
```

Output

```
$ lowentc --run big_doubled short_buffer.low [1,3,4] [0,0]
big_doubled([1,3,4], [6,8]) = 2
  arg0 (written) = [1,3,4]
  arg1 (written) = [6,8]
$ lowentc --run big_doubled short_buffer.low [3,4,5,6] [0,0]
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds on write (panic)
```

From `[1,3,4]` two elements remain (3, 4), and they fit in two slots. `[3,4,5,6]` leaves four, so the run **stops** as the third is stored. It used to end quietly after two, and nobody learned that the other two were gone. When both lengths are known at translation (an `array 5 u8` input,

> say), there is nothing to stop: it is refused with `E-COLLECT-FULL`. A `pipe` does not allocate, so it does not grow the buffer either. To keep only what fits, **write** `take 2`.

24.7 This chapter's syntax at a glance

Shape	Meaning	Why
`pipe xs do … end`	scan the source `xs` once	the language owns the skeleton (index, end test)
`filter over2 . · map dbl .`	keep · transform — pass a named op	no lambdas — the name is the explanation
`take 2 . · skip 1 .`	only the first few · drop the first few	read only as much as needed
`enumerate idxadd . · zip ys addb .`	pass the index or partner as op arguments	no tuples are built
`scan 0 addb . · fold 0 addu .`	emit running values · accumulate into one value	the op takes the accumulator first, then the element
`count . · any is_zero . · all under10 .`	terminators that produce a value	usable as `return pipe … end`
`collect into out .`	store into the caller's buffer	a `pipe` never allocates — a buffer too short is refused or stops the run
exactly one terminator, at the end	a stage after it is `E-PIPE-NO-TERMINAL`	the end of the flow is in one place
a word like `sort`	does not exist — `E-PIPE-STAGE`	operations that cannot fuse were left out

Table 24.2 — `pipe` syntax — shape · meaning · why it looks this way

> **Recap**
>
> `pipe <source> do … end` passes through stages (`filter`, `map`, `take`, `skip`, `enumerate`, `zip`, `scan`) and ends with one terminal (`collect into`, `fold`, `count`, `any`, `all`). Stages take named ops, and the buffer to collect into is given by the caller. Walking once with no intermediate arrays is the definition, and operations that cannot be fused are not in the vocabulary. `any`, `all` and `take` read only as much as needed.

Part VII — Concurrency

25 Actors — living with state, by messages

In this chapter

25.1 Declare, make, talk 313
25.2 Carrying values in messages 314
25.3 Mailboxes — putting in and emptying 316
25.4 Let it crash, then restart 317
25.5 Where actors may be used 319
25.6 Actors that hold a capability, and what state may contain 319
25.7 Designing with actors — a transfer between two accounts 322
25.8 Common mistakes 323
25.9 This chapter's syntax at a glance 330

What to know first

chapter 15, Effects · `state` and `panic` are effects written without capabilities
chapter 19, Ownership · once ownership is handed over, the sender cannot use the value
chapter 20, Allocators and fixed memory · an allocator is an actor satisfying a trait

Looking back

How was chapter 20's bump allocator made, and how was it called? Where did its cursor live?

> **A.** It was made with `spawn actor allocs.bump_bytes` and called with `send a reserve 3`. The cursor lived in the actor's state, and the outside reached that state only through messages. This chapter covers actors properly.

The need for this chapter, and its context

Part VII covers places where several things progress together. Most concurrency defects come from **several flows touching the same state together**. Locks can prevent that, but a forgotten lock or a wrong order is silently wrong. Actors approach from the other side. State is kept only inside the actor, and messages are handled one at a time. Then there is simply no way to touch that state concurrently. Part VII opens with actors because this shape is already used throughout the language, as in the standard library's allocators.

By the end of this chapter

You will learn to declare an actor with `actor … do state do … end … end`, make one with `spawn actor`, and send messages with `send`. You will see that ops inside actors follow the `fn`/`proc` rules too, and how to carry values in messages and hand over ownership. You will also see mailboxes filled with `spawn send` and emptied later with `drain`, restarting crashed actors with `failure restart`, and `build profile`, which decides where actors may be used.

The questions this chapter answers

1 Isn't calling an actor just a function call in the end?

25.1 Declare, make, talk

examples/ch25/counter.low

```
module counter_demo .
rem run: use_counter

actor counter do
  state do
    value u64 .
  end

  proc inc output u64 . effects state .
  do
    set value (add value 1) .
    return value .
  end

  fn get output u64 .
  do
    return value .
  end
end

proc use_counter output u64 . effects state .
do
  var c counter be spawn actor counter .
  let a u64 be send c inc .
  let b u64 be send c inc .
  return send c get .
end
```

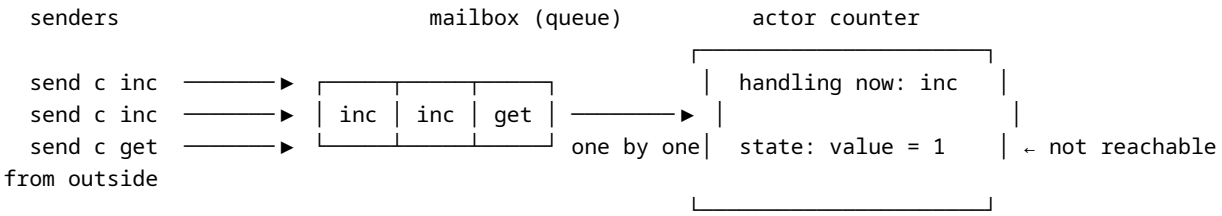
Output

```
$ lowentc --run use_counter counter.low
use_counter() = 2
```

- `actor counter do … end` declares an actor. `state do value u64 . end` is its state, which exists only inside the actor and cannot be touched directly from outside.
- `inc`, which changes the state, is a `proc` with `effects state`. `get`, which only reads, is a `fn`. Ops inside actors follow the kind rules of chapter 5 as is.
- `var c counter be spawn actor counter .` makes one actor. Its state starts at 0.
- `send c inc` sends the `inc` message to `c`, waits until it is handled, and receives the result.

An actor handles messages **one at a time**. So its state is never touched concurrently. That is why there is no need to take locks by hand.

Think of a bank with a single counter. Customers (messages) queue, and the clerk (the actor) serves one customer at a time. The vault (the state) is behind the counter, so no customer can open it directly.



Because messages leave the queue one at a time, two messages never change `value` at the same moment.

Writing `fn` for something that changes state is rejected.

examples/ch25/pure_bad.low

```
module pure_bad .
rem expect: E-EFFECT-PURITY

actor counter do
  state do
    value u64 .
  end

  fn inc output u64 .
  do
    set value (add value 1) .
    return value .
  end
end
```

Output

```
$ lowentc --check pure_bad.low
pure_bad.low:9:0 E-EFFECT-PURITY: a `fn` message handler WRITES the actor's STATE — and
that write is visible to the NEXT message. A fn is an enforced purity contract (SPEC-003
 §27): callers may memoise, reorder or elide it. Declare it a `proc` (and say `effects s
tate`). ★ Nobody used to look here: handlers were declared with `on`, which said NOTHING
 about pure-vs-procedural, and the IR read that as PURE. RFC-0046 removed the bare-`op`
default exactly so the 1 bit is always stated — `on` had put it back. RFC-0057
pure_bad.low:10:3 E-EFFECT-CALC: this fn is declared pure but performs `state` — make it
 a `proc` with `effects …`, or remove the effect
```

As the diagnostic says, that write is visible to **the next message**. If a caller remembered results or reordered calls, the answer would be wrong.

25.2 Carrying values in messages

Values are written side by side after the message name. On the receiving side the actor itself is treated like the first parameter, and the carried values follow.

examples/ch25/args.low

```
module args .
rem run: tally 5 7

actor account do
```

```
  state do
    balance u64 .
  end

  proc deposit input amount u64 . output u64 . effects state .
    requires le amount 1000000 .
  do
    set balance (wrap_add balance amount) .
    return balance .
  end
end

proc tally input a u64 . input b u64 . output u64 . effects state .
  requires le a 1000000 .
  requires le b 1000000 .
do
  var acct account be spawn actor account .
  let x u64 be send acct deposit a .
  return send acct deposit b .
end
```

Output

```
$ lowentc --run tally args.low 5 7
tally(5, 7) = 12
```

`send acct deposit a` puts `a` into `deposit`'s `amount`. Message ops can have contracts too.

Sending a value with ownership hands the ownership over.

examples/ch25/handoff.low

```
module handoff .
rem expect: E-OWN-MOVED

type job u64 .

actor worker do
  state do
    done u64 .
  end
  proc take input j owned job . output u64 . effects state .
  do
    drop j .
    set done (add done 1) .
    return done .
  end
end

proc give_twice input n u64 . output u64 . effects state .
do
  var w worker be spawn actor worker .
  var j owned job be n .
  let a u64 be send w take j .
  return send w take j .
end
```

Output

```
$ lowentc --check handoff.low
handoff.low:23:0 E-OWN-MOVED: this `owned` value was already MOVED (consumed) — using it
 again is use-after-move, which SPEC-004 §4.8 has always called a compile error and whic
h nothing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set <name> <new v
alue>` re-initialises the place — that is how a handle threads through a loop), or borro
w it LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered yet (E-IR-UN
SUP says so at the call site), so `f (ref h)` is not a way out today
```

Once `j` has gone out with the first message, `give_twice` can no longer use `j`. When two flows hold the same value together, races arise. When ownership travels only by message, only one party can touch the value at any moment, and a race is not so much prevented as **left no place to happen**.

> **Q. Isn't calling an actor just a function call in the end?**
>
> **A.** `send` resembles a call in syntax, but two things differ. State is locked inside the actor, so the caller has no way to read or write its fields, and handling one message at a time is guaranteed. Also, holding something borrowed from an actor while talking to that actor again is rejected (`E-BORROW-EXCL`), because we would be looking at the state while the actor changes it.

25.3 Mailboxes — putting in and emptying

`send` waits until handling finishes. Deliveries whose results are not needed are only **put** into the mailbox with `spawn send`.

examples/ch25/mailbox.low

```
module mailbox .
rem run: drive
rem run: nodrain

actor counter do
  state do
    v u64 .
  end
  proc inc output u64 . effects state .
  do
    set v (add v 1) .
    return v .
  end
  fn get output u64 .
  do
    return v .
  end
end

proc drive output u64 . effects state .
do
  var c counter be spawn actor counter .
  spawn send c inc .
  spawn send c inc .
  spawn send c inc .
```

```
  drain c .
  return send c get .
end

proc nodrain output u64 . effects state .
do
  var c counter be spawn actor counter .
  spawn send c inc .
  spawn send c inc .
  return send c get .
end
```

Output

```
$ lowentc --run drive mailbox.low
drive() = 3
$ lowentc --run nodrain mailbox.low
nodrain() = 0
```

`drive` puts `inc` in three times, empties the mailbox in the order put in with `drain c`, and then reads 3. `nodrain` did not empty it, so the result is 0. Messages in a mailbox **are not handled by themselves**.

The processor does not decide delivery times on its own for the sake of **determinism**. The same program must give the same answer, and the delivery time is the answer. Because a person picks where to empty, the VM and native code deliver in the same order, and the cross-check of two back ends keeps it so. `schedule .` empties the mailboxes of every actor at once. Fixing the mailbox size with `mailbox bounded 2 .` makes overflowing puts stop, and `try spawn send` gives the error as a value instead of stopping.

> **A common misconception. Every actor is its own thread**
>
> An actor is a **shape** of execution, not a thread. In this edition's processor, `send`, `spawn send` and `drain` are delivered deterministically within one flow. What actors give is not parallelism but **isolation of state**. Splitting work over several flows is the job of tasks and channels (chapter 26), and computing data split up at the same time is the job of parallel loops (chapter 27).

25.4 Let it crash, then restart

When an actor's op `panic`s, the actor can be restarted. Its state returns to the beginning and the message is handled again.

examples/ch25/restart.low

```
module restart .
rem run: recover
rem trap: escalate

actor recov do
  state do
    bad u64 .
  end
```

```
  failure restart max 3 .
  proc step output u64 . effects state panic .
  do
    if ne bad 0 . do
      panic "corrupted state" .
    end
    set bad 1 .
    return 42 .
  end
end

actor doomed do
  state do
    v u64 .
  end
  failure restart max 2 .
  proc boom output u64 . effects state panic .
  do
    panic "always fails" .
    return 0 .
  end
end

proc recover output u64 . effects state panic .
do
  var c recov be spawn actor recov .
  let x u64 be send c step .
  return send c step .
end

proc escalate output u64 . effects state panic .
do
  var c doomed be spawn actor doomed .
  return send c boom .
end
```

Output

```
$ lowentc --run recover restart.low
recover() = 42
$ lowentc --run escalate restart.low
== ir diagnostics (1) ==
0:0 E-VM-PANIC: the program called `panic` — this is an unrecoverable stop, and it is NO
T a contract violation (the code chose to stop, it did not break a promise)
```

- `recov` sets `bad` to 1 on the first `step` and `panic`s on the second. With `failure restart max 3 .`, the state goes back to the start (`bad = 0`) and the message is handled again, giving 42.
- `doomed` always crashes. If it still crashes after two restarts, the failure **goes upwards** and the program stops. No actor is left silently stopped.

There are three policies — `restart max <count>` (that many times), `never` (pass upwards on the first crash) and `always` (restart without counting). Restarting applies only to `panic`. A contract violation is the program breaking a promise it wrote itself, and doing it again gives the same result, so it is not restarted. Only **the actor's own state** is revived; other actors are not touched.

Code that undoes, one by one, failures where state became strange by chance is long, and long code is itself wrong. Resetting the state to the beginning is a short recovery that is always right.

25.5 Where actors may be used

A program can state where it runs with `build profile <name> .`. When it does, concurrency that place cannot support is rejected at translation.

Profile	Level	Up to
`freestanding`	0	No operating system. No concurrency
`embedded`	1	Running fixed work split up (flows fixed statically)
`native`	2	Making flows and exchanging over channels
`server`	3	Actors

Table 25.1 — Profiles and the levels they open

examples/ch25/profile.low

```
module profile .
rem expect: E-PROFILE-LEVEL

build profile embedded .

actor counter do
  state do
    value u64 .
  end
  proc inc output u64 . effects state .
  do
    set value (add value 1) .
    return value .
  end
end
```

Output

```
$ lowentc --check profile.low
4:0 E-PROFILE-LEVEL: this module uses a CONCURRENCY FEATURE its build profile does not p
rovide (RFC-0009 §105/§139): freestanding has no concurrency at all, embedded has a stat
ic task graph, native adds jobs/green threads, server adds actors. The level is not decl
ared — it is COMPUTED from the constructs you used — and the profile caps it. Either rai
se the profile or stop using the feature
```

Declaring an `actor` alone is level 3. Without a profile, nothing is restricted — only whoever writes it takes on the promise.

25.6 Actors that hold a capability, and what state may contain

An actor's state may have a capability field. A capability is a mark that exists only at translation, so the field has no size at run time. In return, a rule fixes what fills it.

examples/ch25/capfield.low

```
module capfield .
rem run: main

rem a log book that carves 16 bytes from the fixed window for each entry
actor logbook do
  state do
    rem a capability field; it has no size at run time and is filled from the capability
 at the spawn site
    root cap allocator .
    entries u64 .
  end
  proc write input code u8 . output bool . effects alloc state . do
    let g option mut slice u8 . . be alloc_bytes root capacity 16 .
    guard is_some g . else return false .
    let v mut slice u8 . . be some_value g .
    set (index v 0) code .
    set entries (add entries 1) .
    return true .
  end
  fn count output u64 . do
    return entries .
  end
end

proc main input al cap allocator . output u8 . effects alloc state .
do
  var book logbook be spawn actor logbook .
  let a bool be send book write 7 .
  let b bool be send book write 9 .
  return narrow u8 (send book count) .
end
```

Output

```
$ lowentc --run main capfield.low
main() = 2
```

- `root cap allocator .` is the capability field. `write` calls `alloc_bytes root capacity 16` through it, so the capability need not be passed as an argument in every message.
- `spawn actor logbook` is allowed because `main` received `cap allocator`. The field is filled with **the capability at the spawn site**.
- After two entries, `count` answers 2.

An op that did not receive the capability is refused when it spawns the same actor.

examples/ch25/mistake_capfield.low

```
module mistake_capfield .
rem expect: E-CAP-FORGE

actor logbook do
  state do
    root cap allocator .
    entries u64 .
  end
  proc write input code u8 . output bool . effects alloc state . do
```

```
    let g option mut slice u8 . . be alloc_bytes root capacity 16 .
    guard is_some g . else return false .
    set entries (add entries 1) .
    return true .
  end
end

rem ✘ an op that was not handed cap allocator spawns an actor with a capability field
proc sneaky output u64 . effects alloc state .
do
  var book logbook be spawn actor logbook .
  let a bool be send book write 7 .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_capfield.low
mistake_capfield.low:20:0 E-CAP-FORGE: this spawns an actor whose state HOLDS a capabili
ty (`cap allocator` / `cap heap`), but the op spawning it holds no capability of that ki
nd. A capability field means nothing at run time — it is only real because the place tha
t creates the actor already had the right. Without that, one `spawn` would forge authori
ty out of thin air (RFC-0112 D6). Take the capability as an input of this op
```

If this were allowed, one `spawn` line would conjure a capability out of nothing. A capability field only records the fact that "the place that created this actor already held the capability". The standard allocators `allocs.fixed_bytes` and `heap_bytes` are ordinary actors that follow this rule (chapter 20).

This is what a state field may and may not hold.

Type	Accepted?	Why
numbers · `bool` · `option` · structs	yes	they are values, so they stay inside the actor
`slice` · `mut slice`	yes	this is how an allocator holds its backing bytes (`allocs.bump_bytes`)
`cap allocator` · `cap heap`	yes — the spawning op must hold the same capability	`E-CAP-FORGE` stops forging
`array <count> <type>`	refused — `E-TYPE-ARRAY`	fixed-length arrays are accepted only as op inputs; there is nowhere to keep the length
`ref` · `mut_ref`	accepted in this edition — but using it stops the program	there is nowhere to say what it borrows; see "Common mistakes" below

Table 25.2 — Types of actor state fields

25.7 Designing with actors — a transfer between two accounts

This section gathers the pieces so far into one design. Each account is an actor, and a transfer is an op that talks to the two actors in turn.

examples/ch25/transfer.low

```
module transfer .
rem run: move 30
rem run: move 80

enum bank_error do
  insufficient .
end

actor account do
  state do
    balance u64 .
  end

  rem add to the balance --- it changes state, so it is a proc
  proc deposit input amount u64 . output u64 . effects state .
    requires le amount 1000000 .
  do
    set balance (wrap_add balance amount) .
    return balance .
  end

  rem if short, leave the state untouched and return the error as a value
  proc withdraw input amount u64 . output result u64 bank_error . effects state .
    errors insufficient .
  do
    guard le amount balance . else return error insufficient .
    set balance (sub balance amount) .
    return ok balance .
  end

  fn peek_balance output u64 .
  do
    return balance .
  end
end

rem a transfer between two actors --- deposit only when the withdrawal succeeded
proc move input amount u64 . output u64 . effects state .
  requires le amount 1000 .
do
  var src account be spawn actor account .
  var dst account be spawn actor account .
  let seed u64 be send src deposit 50 .
  let r result u64 bank_error be send src withdraw amount .
  guard not (is_error r) . else return add (mul (send src peek_balance) 1000) (send dst
peek_balance) .
  let t u64 be send dst deposit amount .
  return add (mul (send src peek_balance) 1000) (send dst peek_balance) .
end
```

Output

```
$ lowentc --run move transfer.low 30
move(30) = 20030
$ lowentc --run move transfer.low 80
move(80) = 50000
```

- The balance lives only in the state of `account`. The outside reaches an account only through three messages: `deposit`, `withdraw` and `peek_balance`.
- When the balance is short, `withdraw` returns `error insufficient` **without touching the state**. It does not stop — a short balance is a failure the account's user can handle (chapter 17).
- `move` deposits only when the withdrawal succeeded. `move 30` leaves 20 with the sender and 30 with the receiver, giving 20030. With `move 80` the withdrawal fails and both accounts stay at 50 and 0 (50000).
- Messages are processed one at a time, so no other message can slip in between `withdraw` checking the balance and reducing it. The race between "check, then write" does not exist in this shape.

The `errors insufficient .` of `withdraw` has no condition on purpose. The fourth item of "Common mistakes" below explains why.

25.8 Common mistakes

> **Counter-example. Putting messages into a bounded mailbox without draining it**

examples/ch25/mistake_bounded.low

```
module mistake_bounded .
rem trap: burst

actor counter do
  state do
    value u64 .
  end
  mailbox bounded 2 .
  proc inc output u64 . effects state .
  do
    set value (add value 1) .
    return value .
  end
  fn get output u64 .
  do
    return value .
  end
end

proc burst output u64 . effects state .
do
  var c counter be spawn actor counter .
  spawn send c inc .
  spawn send c inc .
  rem ✘ the mailbox holds two, yet a third is put in before draining
```

```
  spawn send c inc .
  drain c .
  return send c get .
end
```

Output

```
$ lowentc --run burst mistake_bounded.low
== ir diagnostics (1) ==
0:0 E-VM-MAILBOX-FULL: the actor's bounded mailbox is full — `spawn send` would exceed `
mailbox bounded N` pending messages (drain it first, or raise N)
```

`mailbox bounded 2 .` promises "at most two pending messages". The third `spawn send` overflows and stops with `E-VM-MAILBOX-FULL`. To handle it instead of stopping, send with `try spawn send`. The overflow then comes back as a `result`, and you can drain and send again.

examples/ch25/bounded_fixed.low

```
module bounded_fixed .
rem run: burst

actor counter do
  state do
    value u64 .
  end
  mailbox bounded 2 .
  proc inc output u64 . effects state .
  do
    set value (add value 1) .
    return value .
  end
  fn get output u64 .
  do
    return value .
  end
end

proc burst output u64 . effects state .
do
  var c counter be spawn actor counter .
  spawn send c inc .
  spawn send c inc .
  rem if it cannot be put in, receive the error as a value instead of stopping --- drain
, then put it in again
  let r result u64 mailbox_full be try spawn send c inc .
  if is_error r . do
    drain c .
    spawn send c inc .
  end
  drain c .
  return send c get .
end
```

Output

```
$ lowentc --run burst bounded_fixed.low
burst() = 3
```

Counter-example. Sending a message the actor does not handle

examples/ch25/mistake_unknownmsg.low

```
module mistake_unknownmsg .
rem expect: E-IR-UNDEF

actor counter do
  state do
    value u64 .
  end
  proc inc output u64 . effects state .
  do
    set value (add value 1) .
    return value .
  end
end

proc use_counter output u64 . effects state .
do
  var c counter be spawn actor counter .
  rem ✘ `counter` does not handle a `dec` message
  return send c dec .
end
```

Output

```
$ lowentc --check mistake_unknownmsg.low
mistake_unknownmsg.low:19:0 E-IR-UNDEF: send names a message this actor does not handle
(the handler is looked up IN THE RECEIVER'S TYPE — it used to be found by BARE NAME, so
two actors with a handler of the same name silently shared one)
```

A message name is looked up **in the receiving actor's type**. `counter` has no `dec`, so this is `E-IR-UNDEF`. As the diagnostic's note tells, lookup was once by bare name, and two actors with a handler of the same name silently shared one.

Counter-example. Reading an actor's state field from outside

examples/ch25/mistake_peekstate.low

```
module mistake_peekstate .
rem expect: E-ACTOR-FIELD

actor counter do
  state do
    value u64 .
  end
  proc inc output u64 . effects state .
  do
    set value (add value 1) .
    return value .
```

```
  end
end

proc peek output u64 . effects state .
do
  var c counter be spawn actor counter .
  let a u64 be send c inc .
  rem ✘ reads the actor's state field from outside --- this should be blocked, but passe
s in this edition
  return field c value .
end
```

Output

```
$ lowentc --check mistake_peekstate.low
mistake_peekstate.low:20:0 E-ACTOR-FIELD: this reads a STATE FIELD of an actor from outs
ide it. An actor's state lives inside the actor and nowhere else (§10.2): the only door
is a message (`send a <op> …`). If the state were readable from outside, the isolation t
hat makes actors safe without locks — one message at a time — would not hold, and the re
ader would see a value between two messages. Add an op to the actor that returns what yo
u need
```

The first promise of this chapter was “the state lives only inside the actor and cannot be touched directly from outside”. `field c value` tries to go around that door and is refused with `E-ACTOR-FIELD`. If state were readable from outside, handling one message at a time would buy nothing — the reader would see a value between two messages. If you need the state, give the actor a read message such as `get` and ask with `send c get`.

Counter-example. Using a state field in the error condition of a message op

examples/ch25/mistake_errorsstate.low

```
module mistake_errorsstate .
rem expect: E-ERRORS-STATE

enum bank_error do
  insufficient .
end

actor account do
  state do
    balance u64 .
  end

  proc deposit input amount u64 . output u64 . effects state .
    requires le amount 1000000 .
  do
    set balance (wrap_add balance amount) .
    return balance .
  end

  rem ✘ the error condition reads the state field `balance` --- on exit it is read again
, after the balance has shrunk
  proc withdraw input amount u64 . output result u64 bank_error . effects state .
```

```
    errors insufficient gt amount balance .
  do
    guard le amount balance . else return error insufficient .
    set balance (sub balance amount) .
    return ok balance .
  end
end

proc take30 output u64 . effects state .
do
  var a account be spawn actor account .
  let seed u64 be send a deposit 50 .
  let r result u64 bank_error be send a withdraw 30 .
  guard not (is_error r) . else return 1 .
  return ok_value r .
end

var fee u64 be 5 .

rem ✘ the same holds outside an actor --- a module `var` and a `mut` argument can both b
e changed by the body, so neither may stand in an error condition
proc charge input amount u64 . input log mut slice u8 . . output result u64 bank_error .
 effects state .
  errors insufficient lt amount fee .
do
  set fee 0 .
  return ok amount .
end
```

Output

```
$ lowentc --check mistake_errorsstate.low
mistake_errorsstate.low:42:0 E-ERRORS-STATE: the condition of this `errors` clause names
 something whose value can DIFFER between entry and exit — an actor state field, a modul
e `var`, or a `mut` parameter. An `errors` condition is read on the values the op was EN
TERED with (canon §6.4.2): the clause says what the CALLER got wrong, and what the calle
r handed over is all it can be blamed for. A name the body may change cannot carry that
meaning — read on exit it accuses the op of owing an error it never owed. Write the cond
ition over the inputs that do not change (and module constants), and guard on the changi
ng thing inside the body
21:0 E-ERRORS-STATE: the condition of this `errors` clause names something whose value c
an DIFFER between entry and exit — an actor state field, a module `var`, or a `mut` para
meter. An `errors` condition is read on the values the op was ENTERED with (canon §6.4.2
): the clause says what the CALLER got wrong, and what the caller handed over is all it
can be blamed for. A name the body may change cannot carry that meaning — read on exit i
t accuses the op of owing an error it never owed. Write the condition over the inputs th
at do not change (and module constants), and guard on the changing thing inside the body
mistake_errorsstate.low:42:0 E-NAME-BUILTIN: a PARAMETER takes the name of a builtin op
— inside this op the name now means two things, and which one it means decides how the s
entence is bracketed (RFC-0046 P1). Rename the parameter
```

`errors insufficient gt amount balance .` was written to mean “this error when withdrawing more than the balance”. But an `errors` condition is read on the values the op was **entered** with (canon 6.4.2). `errors` stands on the same side as `requires` — it says what the **caller** got wrong, and what the caller did is hand things over. So a name the body can change (a state field, a

module `var`, a `mut` parameter) cannot stand in the condition, and translation refuses it with `E-ERRORS-STATE`. Read on exit, a successful run would accuse itself: the balance dropped from 50 to 20, so on exit `gt 30 20` is true, which amounts to "the condition holds, yet the error was not returned". Write the error condition over the **inputs**, and let the `guard` in the body judge the state — as `transfer.low` above does with a bare `errors insufficient ..`

Counter-example. Keeping a borrow in a state field

examples/ch25/mistake_reffield.low

```
module mistake_reffield .
rem expect: E-ACTOR-STATE-REF

actor holder do
  state do
    rem ✘ keeping a borrow in state; there is no place that says what it borrows
    r ref u64 .
  end
  fn peek output u64 . do
    return deref r .
  end
end

proc peek_it output u64 . effects state .
do
  var h holder be spawn actor holder .
  return send h peek .
end
```

Output

```
$ lowentc --check mistake_reffield.low
7:0 E-ACTOR-STATE-REF: an actor STATE field may not be a borrow (`ref` / `mut_ref`). A b
orrow may not outlive what it borrows (§8.4.1), and a state field lives as long as the a
ctor — there is nowhere here to say what it borrows. Keep a VALUE in state (copy it in),
 or keep a slice the actor was handed and owns for its lifetime
mistake_reffield.low:17:0 E-ACTOR-UNINIT: this is the FIRST message sent to an actor tha
t was just spawned, and the handler READS a state field that holds a slice without setti
ng it. A fresh actor's state is all zeroes, and zero is not a slice: the VM stops (`len
needs a slice`) while native quietly answers `none` — the two back ends disagree, which
means one of them is lying. Send the message that sets it up first (the one whose handle
r `set`s that field, typically `init`)
```

A borrow (`ref`) cannot outlive what it borrows (chapter 12). An actor's state stays for as long as the actor lives, so the field has nowhere to say what it borrows. So the declaration is refused with `E-ACTOR-STATE-REF`. Until 2026-09-16 the declaration was accepted and the actor spawned with the field empty; only at `deref r` did the VM stop with `E-VM-TYPE` and native code with a `panic`. Keep values in state instead of borrows, and if a value is large, keep a slice the actor owns for its lifetime.

A common misconception. A restarted actor continues from the state just before it blew up

examples/ch25/restart_resets.low

```
module restart_resets .
rem run: three_incs

actor counter do
  state do
    value u64 .
  end
  failure restart max 3 .
  proc inc output u64 . effects state panic .
  do
    rem blows up once, when the value is 2
    if eq value 2 . do panic "at two" . end
    set value (add value 1) .
    return value .
  end
end

proc three_incs output u64 . effects state panic .
do
  var c counter be spawn actor counter .
  let a u64 be send c inc .
  let b u64 be send c inc .
  rem the third message blows up, the state goes back to 0, and this message is processe
d again
  return send c inc .
end
```

Output

```
$ lowentc --run three_incs restart_resets.low
three_incs() = 1
```

After two `inc`s the value is 2, and the third message blows up. A restart puts the state back to **the beginning** (0) and processes that message again, so the answer is 1, not 3. The state just before the failure may be the very reason for the failure, so it is not trusted. Keep values that must not be lost outside the actor — in another actor or a file — and let the restarted actor read them.

A common misconception. Values of the same actor type share their state

examples/ch25/separate_state.low

```
module separate_state .
rem run: two_counters

actor counter do
  state do
    value u64 .
  end
  proc inc output u64 . effects state .
  do
```

```
    set value (add value 1) .
    return value .
  end
end

proc two_counters output u64 . effects state .
do
  var left counter be spawn actor counter .
  var right counter be spawn actor counter .
  let a u64 be send left inc .
  let b u64 be send left inc .
  rem `right` has its own state --- the result is 1
  return send right inc .
end
```

Output

```
$ lowentc --run two_counters separate_state.low
two_counters() = 1
```

`spawn actor counter` twice makes two states. Incrementing `left` twice leaves `right` counting from 1. An actor type is a blueprint; a new state is made on every `spawn`. If many parties must see the same value, keep **one** actor that holds it and have everyone talk to that actor.

25.9 This chapter's syntax at a glance

Shape	Meaning	Why
`actor counter do state do value u64 . end … end`	declare a unit of execution that encloses state	there is no way to touch the state concurrently
`proc inc … effects state . • fn get …`	a message that changes state · one that only reads	`fn`/`proc` rules unchanged — a `fn` that writes is `E-EFFECT-PURITY`
`var c counter be spawn actor counter .`	create one actor (state starts at 0)	each `spawn` has its own state
`send c inc • send acct deposit a`	send and wait until processed · carry a value	actor first — a message is an op taking the actor as first parameter
`spawn send c inc . • drain c . • schedule .`	put in the mailbox · drain that actor's mailbox · drain all	a person picks the delivery point — determinism
`mailbox bounded 2 . • try spawn send`	mailbox size · receive overflow as a value	overflow stops, or becomes a `result`
`failure restart max 3 . • never • always`	restart a panicked actor from its initial state	when used up, the failure goes upward
`build profile server .`	where actors (level 3) may be used	only those who write it take on the promise

`state do root cap allocator . … end`	a capability field — size 0 at run time	`E-CAP-FORGE` if the spawning op lacks that capability

Table 25.3 — Actor syntax — shape · meaning · why it looks this way

Recap

An actor locks its state inside and handles messages one at a time. It is made with `spawn actor` and called with `send`, which waits, and message ops follow the `fn`/`proc` rules. Messages carry values, and sending an owned value hands over ownership. `spawn send` only puts messages into the mailbox, and `drain` and `schedule` empty it. `failure restart` resets only the state of an actor that `panic`ked, and once exhausted passes the failure upwards. `build profile` decides where actors may be used.

26 Tasks and channels — exchange between bound flows

In this chapter

26.1 Flows are bound to blocks 332
26.2 Channels — containers with an order 334
26.3 Deadlocks you can see from what is written 336
26.4 Flows and memory 337
26.5 What does not exist yet 340
26.6 Common mistakes 341
26.7 This chapter's syntax at a glance 344

What to know first

chapter 15, Effects · `concurrent` brings `wait` with it
chapter 25, Actors · state locked inside, messages handled one at a time
chapter 2, A first program · `test` blocks run under `--test`

Looking back

How were `concurrent` and `wait` told apart in chapter 15?

> **A.** `wait` is waiting that ends eventually without anyone's help (the kernel or a device wakes it), and `concurrent` is waiting that ends only when **another flow of the same program makes progress**. That is why writing `concurrent` counts as writing `wait` too. This chapter covers making those "other flows" and exchanging values between them.

The need for this chapter, and its context

Actors (chapter 25) were the shape that locks state inside. But splitting work over several flows and gathering results, or producers and consumers handing values over, are awkward to write with actors alone. Flow defects come in three common kinds: flows made and forgotten, flows waiting forever for a partner that does not exist, and flows waiting on each other in deadlock. Lowent ties flow lifetimes to blocks to remove the first by grammar, and rejects at translation the second and third when they can be seen from what is written. It also provides a tool that tests **every ordering** of several flows.

By the end of this chapter

You will learn to make flows with `spawn <op>` inside a `task_group` and receive results with `await`. You will see the rule that a flow cannot outlive the block that made it. You will exchange values with `channel`, `chsend` and `chrecv` and understand why channel operations are the `concurrent` effect. You will also see waiting without a partner and deadlocks with no sender rejected at translation, and `schedule explore_interleavings` tests that run every possible ordering.

The questions this chapter answers

1 What does `task_group cancel_on_error` do?

26.1 Flows are bound to blocks

examples/ch26/await.low

```
module awaiting .
rem run: drive
rem test

fn square input n u64 . output u64 .
  requires le n 1000 .
do
  return mul n n .
end

proc drive output u64 . effects state .
do
  var r u64 be 0 .
  task_group do
    var h1 u64 be spawn square 5 .
    var h2 u64 be spawn square 6 .
    let a u64 be await h1 .
    let b u64 be await h2 .
    set r (add a b) .
  end
  return r .
end

test join_any_order schedule explore_interleavings .
do
```

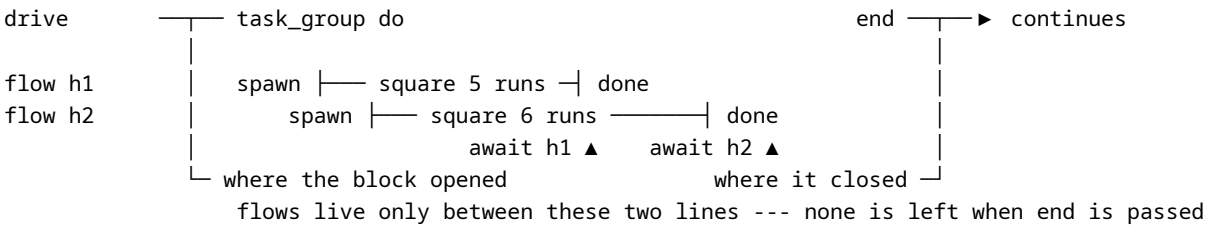
```
  var r u64 be 0 .
  task_group do
    var h1 u64 be spawn square 5 .
    var h2 u64 be spawn square 6 .
    let a u64 be await h1 .
    let b u64 be await h2 .
    set r (add a b) .
  end
  expect eq r 61 .
end
```

Output

```
$ lowentc --run drive await.low
drive() = 61
$ lowentc --test await.low
  [PASS] join_any_order  (schedule: 2 interleavings agree, exhaustive)
== tests: 1 run, 1 passed, 0 FAILED ==
```

- `task_group do … end` binds the flows made inside it into one. Where the block ends, all bound flows end, and none leave with any still running.
- `spawn square 5` makes a flow running `square 5` and gives that flow's **handle**.
- `await h1` waits until that flow ends and gives its result. While waiting, other flows progress.

Drawn on a time axis, the lifetimes look like this.



A flow **cannot outlive the block that made it**. So making a flow outside a binding place is rejected.

examples/ch26/scope.low

```
module scope .
rem expect: E-SPAWN-SCOPE

fn square input n u64 . output u64 .
  requires le n 1000 .
do
  return mul n n .
end

proc loose output u64 . effects state .
do
  var h u64 be spawn square 5 .
  return await h .
end
```

Output

```
$ lowentc --check scope.low
scope.low:12:0 E-SPAWN-SCOPE: spawning a task (`spawn <op>`) is only allowed inside a `t
ask_group` (RFC-0009 D3-b SC1: no task may outlive its group). Wrap it in `task_group do
 … end`, or if you meant an actor instance write `spawn actor <T>`
```

A flow made and forgotten survives after the program ends, touches things already gone, or produces failures nobody reads. When a block holds the lifetime, those three vanish by grammar — there is no place to forget.

`test join_any_order schedule explore_interleavings` in the same file runs **every ordering** of the two flows and checks that the answers agree. “2 interleavings agree, exhaustive” in the result line says so.

> **Q. What does `task_group cancel_on_error` do?**
>
> **A.** When one bound flow ends with an error, siblings that have not started or are waiting are treated as ended with that error. But cancellation **does not unwind**. The effects of flows that already did work are not rolled back; it only stops future work. Which siblings were cancelled depends on the ordering, which the specification does not fix. The error itself shows up through `await`.

26.2 Channels — containers with an order

A channel is a container for putting and taking values between flows. Values come out in the order put in, and the number it can hold is fixed.

examples/ch26/chan.low

```
module chan .
rem run: drive
rem test

actor sink do
  state do
    v u64 .
  end
  proc put input n u64 . output u64 . effects state .
    requires le n 1000 .
  do
    set v (wrap_add v n) .
    return v .
  end
  fn get output u64 .
  do
    return v .
  end
end

proc producer input ch u64 . input n u64 . output u64 . effects concurrent .
do
  return chsend ch n .
end
```

```
proc consumer input ch u64 . input s sink . output u64 . effects state concurrent .
do
  let x u64 be chrecv ch .
  return send s put (min x 1000) .
end

proc drive output u64 . effects state .
do
  var ch u64 be channel u64 .
  var s sink be spawn actor sink .
  task_group do
    spawn consumer ch s .
    spawn consumer ch s .
    spawn producer ch 10 .
    spawn producer ch 32 .
  end
  return send s get .
end

test every_interleaving schedule explore_interleavings .
do
  var ch u64 be channel u64 .
  var s sink be spawn actor sink .
  task_group do
    spawn consumer ch s .
    spawn consumer ch s .
    spawn producer ch 10 .
    spawn producer ch 32 .
  end
  expect eq (send s get) 42 .
end
```

Output

```
$ lowentc --run drive chan.low
drive() = 42
$ lowentc --test chan.low
  [PASS] every_interleaving  (schedule: 58 interleavings agree, exhaustive)
== tests: 1 run, 1 passed, 0 FAILED ==
```

- `channel u64` makes a channel. `chsend ch n` puts in and `chrecv ch` takes out.
- A flow taking from an empty channel stops, and so does a flow putting into a full one. A stopped flow continues when its counterpart comes.
- Two consumers and two producers are bound in one group. Whatever the ordering, both 10 and 32 go into `sink` and the result is 42.

"58 interleavings agree, exhaustive" in the test line means **all** 58 orderings in which the four flows stop and continue were run, and every one answered 42. Concurrency defects usually show only in rare orderings, so tests that run a few times do not catch them. For a small group, running every ordering is the sure way.

Channel operations are the `concurrent` effect, because their completion **depends on another flow's progress**.

examples/ch26/pure_chan.low

```
module pure_chan .
rem expect: E-EFFECT

proc producer input ch u64 . output u64 . effects none .
do
  return chsend ch 1 .
end
```

Output

```
$ lowentc --check pure_chan.low
pure_chan.low:5:1 E-EFFECT: this op performs `concurrent`, `wait`, which its `effects` c
lause does not declare — add it to `effects …`, or stop calling what needs it
```

A producer written as `effects none` calls `chsend` and is rejected for not declaring `concurrent` and `wait`. With this effect in the head, the fact that the op cannot be used on a machine without an operating system — which has no executor to run partner flows — shows at translation time.

26.3 Deadlocks you can see from what is written

A binding place that makes only one flow, where that flow waits for a partner, is rejected.

examples/ch26/alone.low

```
module alone .
rem expect: E-CONC-ALONE

proc consumer input ch u64 . output u64 . effects concurrent .
do
  return chrecv ch .
end

proc lonely output u64 . effects state concurrent .
do
  var ch u64 be channel u64 .
  task_group do
    spawn consumer ch .
  end
  return 0 .
end
```

Output

```
$ lowentc --check alone.low
alone.low:12:0 E-CONC-ALONE: this `task_group` spawns exactly ONE task, and that task ne
eds a PEER to finish (`concurrent` — a `chrecv` on an empty channel, or a `chsend` on a
full one). There is no peer: the parent is waiting on the group. This is not "might be s
low" — NO interleaving completes, so it is a provable deadlock, and the compiler can see
 it here instead of letting it die at runtime. Spawn the other side in the same group, o
r use `wait` I/O, which the KERNEL wakes and which needs no peer (RFC-0022 D-A)
```

Waiting without a partner cannot end in any ordering. It is reported at translation rather than waiting for a hang.

If every bound flow only receives and none sends, it is rejected too.

examples/ch26/deadlock.low

```
module deadlock .
rem expect: E-CONC-DEADLOCK

proc consumer input ch u64 . output u64 . effects concurrent .
do
  return chrecv ch .
end

proc stuck output u64 . effects state .
do
  var ch u64 be channel u64 .
  task_group do
    spawn consumer ch .
    spawn consumer ch .
  end
  return 0 .
end
```

Output

```
$ lowentc --check deadlock.low
deadlock.low:12:0 E-CONC-DEADLOCK: this `task_group`'s tasks wait to RECEIVE (`chrecv`)
but NOTHING sends. There is no `chsend` anywhere in this self-contained module (no `use`
 imports), and the channels are created locally — the enclosing op takes no parameters,
so no caller can feed them either. Every task blocks on an empty channel while the paren
t waits on the group: NO interleaving completes, so it is a PROVABLE deadlock the compil
er names here instead of letting it die at runtime (E-VM-DEADLOCK). Add a producer (`chs
end`), or a task that sends, in the same group (RFC-0022 D-A — the multi-member sibling
of E-CONC-ALONE)
```

> **A common misconception. With the deadlock check, deadlocks cannot happen in this language**
>
> The check catches only deadlocks **visible from what is written**. Two flows waiting crosswise on different channels, or deadlocks that happen only under some condition, are not caught by translation. For those, the processor reports `E-VM-DEADLOCK` at run time when every flow stops, and small cases are found by `explore_interleavings` tests running every ordering. That some deadlocks are not caught makes clear what this rule promises.

26.4 Flows and memory

Tasks run side by side. So two rules apply when a task uses a root from chapter 18 or an allocator from chapter 20.

First, an op that carves from a root cannot be spawned as a task.

examples/ch26/mistake_taskregion.low

```
module mistake_taskregion .
rem expect: E-ALLOC-TASK

proc worker input k u64 . output u64 . effects alloc .
```

```
do
  var n u64 be 0 .
  region r arena do
    let g option mut slice u8 . . be alloc_bytes r capacity 1000 .
    if is_some g . do
      set n k .
    end
  end
  return n .
end

proc two_workers input al cap allocator . output u64 . effects alloc state .
do
  task_group do
    rem ✘ spawning, as a task, an op that carves from a root
    spawn worker 1 .
    spawn worker 2 .
  end
  return 0 .
end
```

Output

```
$ lowentc --check mistake_taskregion.low
20:0 E-ALLOC-TASK: this spawns a TASK whose op takes memory from a ROOT (its effects inc
lude `alloc` or `heap`). A root has one cursor, and regions rewind it in order — two tas
ks interleaving would rewind each other's bytes (measured on the VM's green threads: `re
gion reset needs its own mark`; native tasks are real threads). Give the task an allocat
or over borrowed bytes instead (RFC-0112 D11)
21:0 E-ALLOC-TASK: this spawns a TASK whose op takes memory from a ROOT (its effects inc
lude `alloc` or `heap`). A root has one cursor, and regions rewind it in order — two tas
ks interleaving would rewind each other's bytes (measured on the VM's green threads: `re
gion reset needs its own mark`; native tasks are real threads). Give the task an allocat
or over borrowed bytes instead (RFC-0112 D11)
```

Each root has one cursor, and regions are rewound **in order**. If two tasks open regions in turn, one side's `end` rewinds bytes the other side still uses. A lock cannot fix it, because rewinding relies on order, not on locking. So spawning as a task an op whose effects include `alloc` or `heap` is refused with `E-ALLOC-TASK`.

Second, an allocator whose cursor does not move atomically cannot be handed to a task.

examples/ch26/mistake_tasksshared.low

```
module mistake_tasksshared .
rem expect: E-ALLOC-SHARED

use allocs .

proc user input a allocs.bump_bytes . output u64 . effects state .
do
  let g option mut slice u8 . . be send a reserve 4 .
  if is_some g . do
    return 1 .
  end
  return 0 .
```

```
end

proc share input buf mut slice u8 . output u64 . effects state .
do
  var b allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let z u64 be send b init buf .
  task_group do
    rem ✘ handing an allocator whose cursor is not moved atomically to two tasks
    spawn user b .
    spawn user b .
  end
  return 0 .
end
```

Output

```
$ lowentc --check mistake_taskshared.low
21:0 E-ALLOC-SHARED: the same ALLOCATOR is in play in two places at once here, and its `
reserve` does not move its cursor atomically (no `atomic` effect): it is handed to more
than one task, or handed to a task and still used beside it. Two of them reserving race
on one cursor. Give each task its OWN allocator over its own bytes — two `spawn actor` i
nstances, each `init`ed on a disjoint slice — or use one whose `reserve` is atomic (RFC-
0112 D11)
```

Two tasks reserving from the same `b` race over one cursor. If the allocator's `reserve` does not declare `atomic`, it is `E-ALLOC-SHARED`. What is refused is the **sharing** — handing one allocator to two tasks, or handing it to a task and still using it beside them. Spawning a separate allocator for each task works. Another way is to hand over **bytes** and create the allocator inside the task, which is what follows.

examples/ch26/taskshared_fixed.low

```
module taskshared_fixed .
rem run: share [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

use allocs .

rem the task receives bytes, not an allocator; the allocator is created inside the task
proc user input piece mut slice u8 . . output u64 . effects state .
do
  var a allocs.bump_bytes be spawn actor allocs.bump_bytes .
  let room u64 be send a init piece .
  let g option mut slice u8 . . be send a reserve 4 .
  if is_some g . do
    return send a used .
  end
  return 0 .
end

proc share input buf mut slice u8 . output u64 . effects state .
do
  rem split the buffer into two pieces that do not overlap and give one to each task
  var r u64 be 0 .
  task_group do
    var h1 u64 be spawn user (subslice buf 0 8) .
    var h2 u64 be spawn user (subslice buf 8 16) .
```

```
    let a u64 be await h1 .
    let b u64 be await h2 .
    set r (add a b) .
  end
  return r .
end
```

Output

```
$ lowentc --run share tskshared_fixed.low [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
share([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]) = 8
  arg0 (written) = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

`share` splits the buffer into two pieces that do not overlap and gives one to each task. Each task spawns an allocator on its own piece and uses 4 bytes; adding the two answers gives 8. Because the pieces do not overlap, there is no cursor to race over and no bytes to be rewound.

26.5 What does not exist yet

Channels without an end (`channel … unbounded`) and lock state shared by several flows (`lock`, `rwlock`) are not accepted yet (`E-CHAN-UNBOUNDED`, `E-LOCK-NOTYET`). One could accept what does not exist and build it later, but then programs written meanwhile would find that what seemed to work does not. It is more honest to say it does not exist. Handling the same memory atomically from several flows is covered in chapter 27.

examples/ch26/lock.low

```
module lock .
rem expect: E-LOCK-NOTYET

rem ✘ lock state shared between flows; the name is in the canon but it is not accepted y
et
fn read_count input c lock u64 . output u64 . do
  return 0 .
end
```

Output

```
$ lowentc --check lock.low
lock.low:5:0 E-LOCK-NOTYET: `lock t` / `rwlock t` / `shared_read t` are level-3 shared-s
tate types that SPEC-008 §74 names but this compiler DOES NOT IMPLEMENT: no lock is take
n, no release happens. Writing one tells every reader the state is lock-protected while
NOTHING checks that — an unchecked safety claim is the lie PRINCIPLES.md §0 is about. ★
And today there is nothing for a lock to protect against: execution is cooperative on ON
E OS thread and actor delivery is SEQUENTIAL, so such a lock could never even be exercis
ed. This is refused for the same reason as `mailbox unbounded` and `bounded 0`. Use an `
actor` (sequential delivery IS the mutual exclusion) or the `atomic_*` ops, whose memory
 ordering IS real (RFC-0018 §6.1). These types come back when real OS threads do
lock.low:5:0 W-NOT-YET: this type NAME is accepted but carries NO MEANING yet: lowering
cannot read it, so every op in its signature falls to the interpreter (~80x). The answer
 stays right, so no oracle will ever see this. Declare it (`type str slice u8 .`) or wri
te the underlying type
```

`lock u64` is a type named in the canon, so the tool does not call it “a type that does not exist”. Instead `E-LOCK-NOTYET` says “not built yet”. Until then, when a count must be shared, use atomic operations (chapter 27) or an actor (chapter 25).

26.6 Common mistakes

Counter-example. Two flows that both start by receiving from each other

examples/ch26/mistake_crossed.low

```
module mistake_crossed .
rem trap: crossed

proc relay input inbox u64 . input outbox u64 . output u64 . effects concurrent .
do
  let x u64 be chrecv inbox .
  return chsend outbox x .
end
proc crossed output u64 . effects state .
do
  var a u64 be channel u64 .
  var b u64 be channel u64 .
  task_group do
    rem ✘ both start by receiving --- no flow puts in the first value
    spawn relay a b .
    spawn relay b a .
  end
  return 0 .
end
```

Output

```
$ lowentc --run crossed mistake_crossed.low
== ir diagnostics (1) ==
0:0 E-VM-DEADLOCK: the scheduler is stuck — every remaining task is blocked on a channel
 (a `chrecv` on an empty channel or `chsend` on a full one) and none can wake another. T
his is a real deadlock, not a tool limit: no interleaving completes
```

`relay a b` receives from `a` and passes to `b`; `relay b a` does the opposite. Both start by receiving, so neither can put in the first value. With two flows and crossed channels, the source alone does not show a deadlock, so translation accepts it. At run time the processor sees every flow blocked and reports `E-VM-DEADLOCK`. To avoid crossed waits, decide for each flow “who sends first”, and where possible let values flow in one direction only.

Counter-example. Lending the same variable for writing to two flows

examples/ch26/mistake_sharedvar.low

```
module mistake_sharedvar .
rem expect: E-EXCL

proc bump input p mut_ref u64 . output void . effects state .
do
```

```
  set p (add (deref p) 1) .
end
proc race output u64 . effects state .
do
  var n u64 be 0 .
  task_group do
    rem ✘ the same variable is lent for writing to two flows
    spawn bump (mut_ref n) .
    spawn bump (mut_ref n) .
  end
  return n .
end
```

Output

```
$ lowentc --check mistake_sharedvar.low
14:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
```

Two flows incrementing the same `n` at once is a data race. The rule that there is only one write borrow (chapter 12) holds between flows too, so it is rejected with `E-EXCL`. The `W-EFFECT-OVER` on the first line is an unrelated defect of this edition (a write through `mut_ref` is not counted as `state`). Let each flow **return** its share, and have the grouping side combine them after `await`.

examples/ch26/sharedvar_fixed.low

```
module sharedvar_fixed .
rem run: count2

rem each flow returns its share, and the grouping side combines them after `await`
fn one output u64 .
do
  return 1 .
end

proc count2 output u64 . effects state .
do
  var n u64 be 0 .
  task_group do
    var h1 u64 be spawn one .
    var h2 u64 be spawn one .
    let a u64 be await h1 .
    let b u64 be await h2 .
    set n (add a b) .
  end
  return n .
end
```

Output

```
$ lowentc --run count2 sharedvar_fixed.low
count2() = 2
```

A common misconception. Concurrent code that gave the right answer once is correct

examples/ch26/order_dependent.low

```
module order_dependent .
rem run: drive
rem test-fail

actor first_seen do
  state do
    v u64 .
  end
  proc put input n u64 . output u64 . effects state .
    requires le n 1000 .
  do
    rem keeps only the first value received --- the answer depends on arrival order
    if eq v 0 . do set v n . end
    return v .
  end
  fn get output u64 .
  do
    return v .
  end
end
proc producer input ch u64 . input n u64 . output u64 . effects concurrent .
do
  return chsend ch n .
end
proc consumer input ch u64 . input s first_seen . output u64 . effects state concurrent
.
do
  let x u64 be chrecv ch .
  let y u64 be chrecv ch .
  let a u64 be send s put (min x 1000) .
  return send s put (min y 1000) .
end
proc drive output u64 . effects state .
do
  var ch u64 be channel u64 .
  var s first_seen be spawn actor first_seen .
  task_group do
    spawn consumer ch s .
    spawn producer ch 10 .
    spawn producer ch 32 .
  end
  return send s get .
end
test first_is_ten schedule explore_interleavings .
do
  var ch u64 be channel u64 .
  var s first_seen be spawn actor first_seen .
  task_group do
    spawn consumer ch s .
    spawn producer ch 10 .
    spawn producer ch 32 .
  end
```

```
  expect eq (send s get) 10 .
end
```

Output

```
$ lowentc --run drive order_dependent.low
drive() = 32
$ lowentc --test order_dependent.low
  [FAIL] first_is_ten
         E-SCHED-NONDET: this test PASSES under some message-delivery orders and FAILS u
nder others (diverged at interleaving 1 of 8 explored) — its result DEPENDS on the order
 the scheduler delivers messages. A deterministic actor program must give the SAME resul
t under EVERY interleaving (RFC-0009 D6 — the actor analogue of a data race). Fix the ha
ndler so order does not matter, or serialize with `drain`
== tests: 1 run, 0 passed, 1 FAILED ==
```

`first_seen` keeps only the first value it receives. Running `drive` gives 32; another order would give 10. One run gives the answer for one order only. The `schedule explore_interleavings` test runs all eight orders and reports with `E-SCHED-NONDET` that the answer depends on the order. A deterministic program must give the same answer in every order. Write handlers whose result does not depend on order, like the `sink` of `chan.low`, which adds up every value it receives.

26.7 This chapter's syntax at a glance

Shape	Meaning	Why
`task_group do … end`	bind flows to a block — all finish at its end	forgotten flows vanish by grammar
`var h1 u64 be spawn square 5 .`	start a flow and receive its handle	outside a group: `E-SPAWN-SCOPE`
`await h1`	wait for that flow's result	other flows progress meanwhile
`task_group cancel_on_error do … end`	cancel siblings on the first error	cancellation does not roll back
`var ch u64 be channel u64 .`	a bounded, ordered container	no unbounded channels yet — `E-CHAN-UNBOUNDED`
`chsend ch n` · `chrecv ch`	put · take — blocks when full or empty	`concurrent` effect — completion depends on others
a lone wait · receives with no sender	rejected at translation (`E-CONC-ALONE` · `E-CONC-DEADLOCK`)	deadlocks visible from the source
`test … schedule explore_interleavings . do … end`	run every possible order and compare answers	tests find bugs of rare orders
`alloc` or `heap` in a task op's effects	`E-ALLOC-TASK`	a root's rewind relies on order — a lock cannot protect it

a non-atomic allocator as a `spawn` argument	`E-ALLOC-SHARED`	hand over byte pieces and create the allocator inside the task
`lock t` · `rwlock t` · `shared_read t`	`E-LOCK-NOTYET`	the name is in the canon — what is not built is not accepted

Table 26.1 — Task and channel syntax — shape · meaning · why it looks this way

Recap

Inside a `task_group`, `spawn <op>` makes a flow and gives a handle, and `await` receives the result. A flow cannot outlive the block that made it, so `spawn` outside a binding place is rejected. `channel`, `chsend` and `chrecv` form an ordered container of fixed size, and channel operations are the `concurrent` effect. Waiting without a partner and deadlocks with no sender are rejected at translation, and `schedule explore_interleavings` tests run every ordering and compare answers.

27 Parallel loops and atomic operations

In this chapter

27.1 Declaring that a loop may be split 346
27.2 The three conditions the processor checks 348
27.3 The combining operation must be associative 349
27.4 Atomic operations 350
27.5 Memory orderings 351
27.6 Lanes — computing several values at once 352
27.7 Saying how a place is used — `access` 356
27.8 Common mistakes 357
27.9 This chapter's syntax at a glance 361

What to know first

chapter 12, Borrowing · many readers or one writer

chapter 26, Tasks and channels · flows are bound to blocks and every ordering can be tested

chapter 16, Capabilities · `effects atomic` pairs with `cap atomic`

Looking back

What did chapter 12 say the exclusivity rule (“many readers or one writer”) has to do with multithreaded programs?

A. That once the rule holds within one flow, the absence of data races follows when the work is split across several flows. This chapter actually uses that property — it splits one loop into pieces run by several flows together, and has translation confirm that splitting gives the same answer.

The need for this chapter, and its context

Splitting the same computation over several cores is the most common parallelisation and the most common place for defects. If the pieces read each other’s elements, accumulate into the same variable, or use operations whose answer depends on order, the program becomes “fast but sometimes wrong”. Lowent **does not trust a declaration that splitting is possible; it checks it**. If it cannot check, it does not quietly run sequentially — it refuses translation. And for the rare places where several flows really must touch the same memory, atomic operations are chosen by name. This is the last chapter of Part VII.

By the end of this chapter

You will learn to declare in an op head that a loop may be split with `parallel <slice> split .`, and the three conditions the processor checks (read only your own share, write only your own share, do not write places that live across iterations). You will state accumulation with `reduce <place> <op> .` and see why non-associative operations are rejected. You will also see atomic operations such as `atomic_add` and `atomic_load`, memory orderings (`order`), and the rules for combining them.

The questions this chapter answers

1 If there are atomic operations, why is there no lock?

27.1 Declaring that a loop may be split

examples/ch27/split.low

```
module split .
rem run: double_all [1,2,3,4,5,6]
rem run: total [1,2,3,4,5,6]

proc double_all input s mut slice u8 . output u64 . effects none .
  parallel s split .
do
  var i u64 be 0 .
  while lt i (len s) . do
    set (index s i) (wrap_mul (index s i) 2) .
    set i (add i 1) .
  end
```

```
  return len s .
end

fn total input s slice u8 . output u64 .
  parallel s split .
  reduce acc add .
do
  var acc u64 be 0 .
  var i u64 be 0 .
  while lt i (len s) . do
    set acc (add acc (widen u64 (index s i))) .
    set i (add i 1) .
  end
  return acc .
end
```

Output

```
$ lowentc --run double_all split.low [1,2,3,4,5,6]
double_all([2,4,6,8,10,12]) = 6
  arg0 (written) = [2,4,6,8,10,12]
$ lowentc --run total split.low [1,2,3,4,5,6]
total([1,2,3,4,5,6]) = 21
  arg0 (written) = [1,2,3,4,5,6]
```

- `double_all`'s head has `parallel s split .` — a declaration that "this loop may be split into pieces of `s` run by several together". Each step reads and writes only `index s i`, its own element.
- `total` accumulates a sum. The accumulator `acc` lives across steps, so as is it cannot be split. `reduce acc add .` states "accumulate per piece, then combine with `add`".

In pictures, this is what the two declarations allow.

```
 split --- each piece touches only its own share
   s:  [ 0 1 2 3 | 4 5 6 7 | 8 9 10 11 ]
         piece A    piece B    piece C        ← all three may run at once without touching each
other

 reduce acc add --- gather per piece, then combine
   piece A: acc_A = 0+1+2+3      = 6  ─┐
   piece B: acc_B = 4+5+6+7      = 22 ─┼─ add ─► acc = 66
   piece C: acc_C = 8+9+10+11    = 38 ─┘
```

`add` gives the same answer however it is grouped (it is associative), so any split matches the sequential result. Floating-point addition is not, so it cannot be split with `reduce` (see "The combining operation must be associative" below).

When the processor confirms such a declaration, it reports `W-PAR-OK`, and that note contains something important: the VM still runs sequentially. That is **a correct implementation** because a theorem is proven that, under the splittable conditions, the parallel result is bit-for-bit identical to the sequential result (chapter 45). Native code really does split into pieces and run them on several threads. The two back ends agreeing is a measurement of that theorem.

27.2 The three conditions the processor checks

Condition	When broken	What goes wrong
Read only your own share	`E-PAR-READ`	Reading another's place makes old-or-new depend on who runs first
Write only your own share	`E-PAR-WRITE`	Two steps writing the same place leave a value that depends on order
Do not write places that live across steps	`E-PAR-CARRY`	Such places tie steps together; to gather, state it with `reduce`

Table 27.1 — Conditions for a loop that may be split

examples/ch27/par_read.low

```
module par_read .
rem expect: E-PAR-READ

proc relative input s mut slice u8 . output u64 . effects none .
  parallel s split .
do
  var i u64 be 0 .
  while lt i (len s) . do
    set (index s i) (wrap_sub (index s i) (index s 0)) .
    set i (add i 1) .
  end
  return len s .
end
```

Output

```
$ lowentc --check par_read.low
par_read.low:9:0 E-PAR-READ: a splittable loop may only read its OWN element of the spli
t slice — reading another index creates a cross-iteration dependence (Bernstein: rd ∩ wr
 = ∅)
```

This loop subtracts the first element from every element. If the first piece changes `index s 0` first, the first element other pieces read has already changed. Even run sequentially, the loop has the defect of subtracting 0 after the first step, and splitting makes that defect depend on ordering.

examples/ch27/par_carry.low

```
module par_carry .
rem expect: E-PAR-CARRY

fn total input s slice u8 . output u64 .
  parallel s split .
do
  var acc u64 be 0 .
  var i u64 be 0 .
  while lt i (len s) . do
    set acc (add acc (widen u64 (index s i))) .
    set i (add i 1) .
  end
```

```
  return acc .
end
```

Output

```
$ lowentc --check par_carry.low
par_carry.low:10:0 E-PAR-CARRY: a splittable loop may not write a local that lives acros
s iterations (a loop-carried dependence) — declare it: `reduce <acc> <op> .`
```

It accumulated into `acc` without `reduce`. The diagnostic says exactly how to fix it.

> **A common misconception. If answers sometimes differ when run in parallel, that is a performance tuning issue**
>
> A program whose answer depends on ordering is not slow but wrong. In Lowent the `parallel` clause does not change meaning. The answer run split and the answer run one by one must always be the same, and if that condition cannot be confirmed, translation refuses. “Fast but sometimes wrong” is not something this language sells.

27.3 The combining operation must be associative

When gathering with `reduce`, the shape of the tree that combines pieces changes depending on how they were split. If the operation is associative, the shape does not change the answer; if not, it does.

examples/ch27/par_assoc.low

```
module par_assoc .
rem expect: E-PAR-ASSOC

fn diff input s slice u8 . output u64 .
  parallel s split .
  reduce acc sub .
do
  var acc u64 be 1000 .
  var i u64 be 0 .
  while lt i (len s) . do
    set acc (sub acc (widen u64 (index s i))) .
    set i (add i 1) .
  end
  return acc .
end
```

Output

```
$ lowentc --check par_assoc.low
par_assoc.low:4:0 E-PAR-ASSOC: this reduction operator is not associative, so the shape
of the reduction tree changes the result — a split tree is not deterministic (★ nonassoc
_shape_matters, Qed — docs/proofs/coq/LowentPar.v)
par_assoc.low:4:0 W-PAR-OK: this loop satisfies the Bernstein conditions and may be spli
t (DET-1 proves the parallel result is bit-identical to the sequential one — docs/proofs
/coq/LowentPar.v). Execution is still sequential, which is a VALID implementation precis
ely because of that theorem
```

`sub` is not associative: `(1000 - 1) - 2` and `1000 - (1 - 2)` differ. Both propositions — associative means the shape does not matter, non-associative means the shape changes the result — are proven in Coq, and the diagnostic cites the theorem's name. Floating-point addition also depends on order, so it is blocked separately (`E-PAR-FLOAT`).

27.4 Atomic operations

Sometimes split pieces must update **one place** together, such as adding counts from each piece into a shared counter. With plain `add`, two threads read the same value, add separately and write, and one addition is lost. An atomic operation is indivisible — other flows cannot see its middle.

examples/ch27/counter.low

```
module atomic_counter .
rem run: main

proc count_par input k cap atomic . input data slice u8 . input counter mut slice u64 .
  output void .
  effects atomic .
  parallel data split .
do
  var i u64 be 0 .
  while lt i (len data) . do
    atomic_add counter 0 1 .
    set i (add i 1) .
  end
  return .
end

proc main input k cap atomic . input al cap allocator . output u8 . effects atomic alloc
 .
do
  let g option mut slice u8 be alloc_bytes al capacity 8 .
  guard is_some g . else return 1 .
  var cells mut slice u64 be view_array u64 (some_value g) .
  atomic_store cells 0 0 .
  count_par k "abcdefghij" cells .
  return narrow u8 (atomic_load cells 0) .
end
```

Output

```
$ lowentc --run main counter.low
main() = 10
```

- `atomic_add counter 0 1` atomically adds 1 at position 0 of the slice `counter`. A place is addressed by **slice and index**.
- `count_par` splits `data` and updates the shared counter. It counted ten bytes, so the result is 10.
- Atomic operations are the `atomic` effect and require receiving `cap atomic`. `main` receives `cap atomic` at the entry point.
- `view_array u64` views the 8 allocated bytes as a slice of one `u64` without copying (chapter 13).

Declaring the atomic effect without the capability is rejected.

examples/ch27/nocap.low

```
module nocap .
rem expect: E-ATOMIC-NOCAP

proc bump input s mut slice u64 . output u64 . effects atomic .
do
  atomic_add s 0 1 .
  return atomic_load s 0 .
end
```

Output

```
$ lowentc --check nocap.low
nocap.low:4:0 E-ATOMIC-NOCAP: this op declares the `atomic` effect but receives NO `cap
atomic`. Like `io`, `alloc` and `heap`, `atomic` is an effect you are HANDED the right t
o: `input k cap atomic .` (§7.2 (6))
```

Atomic operations are not free. Used on a value only one flow touches, they only slow things down. So they are not the default; they are chosen by name, and their cost is written in the head's effects and capabilities.

> **Q. If there are atomic operations, why is there no lock?**
>
> **A.** Lock state shared between flows is not built in this edition, and using it says `E-LOCK-NOTYET`. Instead the standard library has data structures built on atomic operations. `spsc` is a ring buffer through which one producer and one consumer pass values without locks, and its correctness was confirmed by borrowing a proof in a weak-memory model (chapters 34 and 47).

27.5 Memory orderings

Appending `order <name>` to an atomic operation decides what other flows see and when.

Name	Meaning
`seq_cst`	Every flow sees one single order. The strongest; the default when not written
`acq_rel`	For read-write operations, nothing leaks forward or backward
`acquire`	Work after this read does not leak ahead of it
`release`	Work before this write does not leak behind it
`relaxed`	Atomicity only, no ordering

Table 27.2 — Memory orderings

The strongest is the default because it is the easiest to reason about. It is proven that when every access is `seq_cst` you may think sequentially. Weakening is a choice made in writing by someone who knows its value.

Combinations that mean nothing for an operation are rejected.

Using four of the orders with the operations they pair with looks like this.

examples/ch27/orders.low

```
module orders .
rem run: main

rem without an order it is seq_cst; every flow sees one and the same order
rem acq_rel for an operation that reads and writes (atomic_swap), release for a write, a
cquire for a read
proc main input k cap atomic . input al cap allocator . output u8 . effects atomic alloc
 .
do
  let g option mut slice u8 be alloc_bytes al capacity 8 .
  guard is_some g . else return 250 .
  var cells mut slice u64 be view_array u64 (some_value g) .
  atomic_store cells 0 5 order seq_cst .
  let old u64 be atomic_swap cells 0 7 order acq_rel .
  atomic_store cells 0 (add old 10) order release .
  return narrow u8 (atomic_load cells 0 order acquire) .
end
```

Output

```
$ lowentc --run main orders.low
main() = 15
```

`atomic_store … order seq_cst` writes 5, `atomic_swap … order acq_rel`, which reads and writes, swaps in 7 and returns the old 5, `order release` writes 15, and `order acquire` reads it back, so 15 is returned. A pairing that does not match is refused.

examples/ch27/order_bad.low

```
module order_bad .
rem expect: E-ATOMIC-ORDER

proc peek input k cap atomic . input s mut slice u64 . output u64 . effects atomic .
do
  return atomic_load s 0 order release .
end
```

Output

```
$ lowentc --check order_bad.low
order_bad.low:6:0 E-ATOMIC-ORDER: this memory ordering is not valid for this atomic op (
a load cannot be `release`, a store cannot be `acquire`, a fence cannot be `relaxed`) —
RFC-0018 §6.1
```

What would a read release? C leaves such combinations undefined. A write cannot be `acquire`, and a fence (`atomic_fence`), which only sets order, cannot be `relaxed`.

27.6 Lanes — computing several values at once

Where `parallel` splits a loop across **flows**, `vec` holds **several values** as one value within a single flow and computes them at once (SIMD). `vec u32 4` is a value with four lanes holding four `u32`s, and the lane count is part of the type. Whether the machine computes the lanes at once or one by one, the answer is the same (canon 6.2.11).

examples/ch27/lanes.low

```
module lanes .
rem run: capped_sum [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
rem run: extremes [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
rem run: turned [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0] [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
rem run: lanes_here

rem read four lanes as one value, press lanes above 5 down to 5, and add them all
proc capped_sum input b slice u8 . output u32 . effects none .
  requires ge (len b) 16 .
do
  var xs slice u32 be view_array u32 b .
  var v vec u32 4 be load xs 0 .
  var lim vec u32 4 be splat 5 .
  var over mask 4 be gt v lim .
  var capped vec u32 4 be select over lim v .
  return reduce_add capped .
end

rem across the lanes: the largest, the smallest, and the product of all
proc extremes input b slice u8 . output u32 . effects none .
  requires ge (len b) 16 .
do
  var xs slice u32 be view_array u32 b .
  var v vec u32 4 be load xs 0 .
  return add (mul (reduce_max v) 1000) (add (mul (reduce_min v) 100) (reduce_mul v)) .
end

rem reverse the lanes, rotate by one, and write them to memory
proc turned input b slice u8 . input out mut slice u8 . output u32 . effects none .
  requires ge (len b) 16 .
  requires ge (len out) 16 .
do
  var xs slice u32 be view_array u32 b .
  var ys mut slice u32 be view_array u32 out .
  var v vec u32 4 be load xs 0 .
  var r vec u32 4 be reverse v .
  var t vec u32 4 be rotate r 1 .
  store ys 0 t .
  return reduce_add t .
end

rem how many u32 lanes this machine handles at once --- a translation-time number
fn lanes_here output u64 .
do
  return native_lanes u32 .
end
```

Output

```
$ lowentc --run capped_sum lanes.low [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
capped_sum([1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]) = 14
  arg0 (written) = [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
$ lowentc --run extremes lanes.low [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
extremes([1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]) = 9289
  arg0 (written) = [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
```

```
$ lowentc --run turned lanes.low [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0] [0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0]
turned([1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0], [3,0,0,0,9,0,0,0,1,0,0,0,7,0,0,0]) = 20
  arg0 (written) = [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
  arg1 (written) = [3,0,0,0,9,0,0,0,1,0,0,0,7,0,0,0]
$ lowentc --run lanes_here lanes.low
lanes_here() = 4
```

- `load xs 0` reads four lanes starting at position 0 of the slice. `store ys 0 t` writes the other way.
- `splat 5` fills every lane with 5. The lane count comes from the type of the name it is stored in (`vec u32 4`).
- Comparing `vec`s, as in `gt v lim`, gives a **mask** `mask 4` holding true or false per lane. `select over lim v` picks `lim` in lanes where the mask is on and `v` where it is off. Choosing per lane without a branch (`if`) lets the machine do it in one instruction.
- `reduce_add`, `reduce_max`, `reduce_min` and `reduce_mul` gather the lanes into one. Pressing `[1,9,3,7]` down to 5 gives `[1,5,3,5]`, whose sum is 14.
- `reverse` reverses the order of the lanes, and `rotate r 1` rotates them by one. `[7,3,9,1]` rotated, `[3,9,1,7]`, was written to memory.
- `native_lanes u32` gives, **at translation time**, how many `u32` lanes this machine handles at once. Choosing that lane count only computes more at once; the answer is the same.

A mask can also read or write just some of the lanes.

examples/ch27/masked.low

```
module masked .
rem run: keep_big [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0] [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
rem run: read_big [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]

rem write only the lanes above 5 --- places of masked-out lanes are left untouched
proc keep_big input b slice u8 . input out mut slice u8 . output u64 . effects none .
  requires ge (len b) 16 .
  requires ge (len out) 16 .
do
  var xs slice u32 be view_array u32 b .
  var v vec u32 4 be load xs 0 .
  var lim vec u32 4 be splat 5 .
  var m mask 4 be gt v lim .
  store_masked out 0 v m .
  return 0 .
end

rem read only the lanes that are on, and put the default 100 in the others
proc read_big input b slice u8 . output u32 . effects none .
  requires ge (len b) 16 .
do
  var xs slice u32 be view_array u32 b .
  var v vec u32 4 be load xs 0 .
  var lim vec u32 4 be splat 5 .
  var m mask 4 be gt v lim .
  var fallback vec u32 4 be splat 100 .
  var r vec u32 4 be load_masked xs 0 m fallback .
  return reduce_add r .
end
```

Output

```
$ lowentc --run keep_big masked.low [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0] [0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0]
keep_big([1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0], [0,0,0,0,9,0,0,0,0,0,0,0,7,0,0,0]) = 0
  arg0 (written) = [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
  arg1 (written) = [0,0,0,0,9,0,0,0,0,0,0,0,7,0,0,0]
$ lowentc --run read_big masked.low [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
read_big([1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]) = 216
  arg0 (written) = [1,0,0,0,9,0,0,0,3,0,0,0,7,0,0,0]
```

`store_masked out 0 v m` writes only at the places of the lanes that are on (9 and 7) and leaves the others untouched. `load_masked xs 0 m fallback` reads only the lanes that are on and puts the default 100 in the others. This is the shape for handling the tail when fewer than four slots remain at the end of a slice.

Op	What it does	Note
`load` · `store` · `load_masked` · `store_masked`	read and write between memory and lanes	masked forms touch only lanes that are on
`splat` · `select`	one value into every lane · choose per lane by mask	`splat` needs a type context (below)
`reduce_add` · `reduce_max` · `reduce_min` · `reduce_mul`	gather the lanes into one	the result has the element type
`reverse` · `rotate`	reverse · rotate the lanes	the count is a translation-time constant
`native_lanes`	this machine's lane count (translation time)	does not change answers
`sum_neumaier` · `sum_seq`	add up a float slice seen through `view_array`	not lane ops — the name says how they add (compensated · front to back)
`avg`	rounding average per lane	its value is fixed at `(a+b+1)>>1` — the sum is widened so a lane cannot overflow
`prefetch xs i`	pull a place about to be used into cache	a performance hint that does not change meaning

Table 27.3 — Builtin ops for lanes and arrays

Watch two things. First, `splat` takes its lane count from the **declared type**, so it cannot be written inline in an expression. Second, **adding lanes and adding a slice are different ops** — lanes are `reduce_add`, a float slice is `sum_neumaier` or `sum_seq`. Until 2026-09-17 the latter were spelled `sum` and `sum_fast`, while the canon used those same names for "add all the lanes". One name meant two things, so the names were split (canon 6.3.7.1).

Counter-example. Writing `splat` inline in an expression

`examples/ch27/mistake_splatinline.low`

```
module mistake_splatinline .
rem expect: E-VEC-SPLAT

fn capped input b slice u8 . output u32 .
do
  var xs slice u32 be view_array u32 b .
  var v vec u32 4 be load xs 0 .
  rem ✘ `splat` written inline --- nothing says how many lanes
  var over mask 4 be gt v (splat 5) .
  return reduce_add (select over v v) .
end
```

Output

```
$ lowentc --check mistake_splatinline.low
mistake_splatinline.low:9:0 E-VEC-SPLAT: `splat` fills every lane of a vector, and how m
any lanes there are comes from the declared type — inside an expression there is nothing
 to say it, so the value is read as a plain scalar and the surrounding comparison stops
matching its `mask` type. Bind it first, with the lane count written down: `var lim vec
u32 4 be splat 5 .`, then use `lim`
mistake_splatinline.low:9:0 E-TYPE-VAR: the initializer's type does not match the declar
ed type — expected `mask`, found `bool`
```

`splat` fills every lane with one value, and only the declared type says **how many lanes there are**. Inside an expression there is nothing to say it, so the value is read as a scalar and the surrounding comparison stops matching its `mask` type. It is rejected with `E-VEC-SPLAT`; store it first under a name that writes the lane count down — `var lim vec u32 4 be splat 5 .`, then `gt v lim`.

27.7 Saying how a place is used — `access`

The `access <name> <mode> .` clause writes in the head whether the op only **reads** an input place or only **writes** it. Callers and the scheduler rely on that promise.

examples/ch27/access.low

```
module access .
rem run: peek [9,8,7]

rem access data shared_read says this op only reads data, a promise that several tasks m
ay hold it together
fn peek input data slice u8 . output u64 .
  access data shared_read .
do
  return widen u64 (index data 0) .
end
```

Output

```
$ lowentc --run peek access.low [9,8,7]
peek([9,8,7]) = 9
  arg0 (written) = [9,8,7]
```

`shared_read` means “only reads”. With no write there is no race, so several tasks may hold the same place together. `write_only` means “only writes”, so a buffer not yet filled may be passed. The tool **checks** both modes against the body. The remaining modes such as `sequential` are kernel scheduling hints that constrain nothing yet, and writing one makes `W-NOT-YET` say so.

27.8 Common mistakes

Counter-example. Pieces of a split loop incrementing a shared counter with ordinary arithmetic

examples/ch27/mistake_sharedwrite.low

```
module mistake_sharedwrite .
rem expect: E-PAR-WRITE

proc count_par input data slice u8 . input counter mut slice u64 . output void . effects
 none .
  parallel data split .
do
  var i u64 be 0 .
  while lt i (len data) . do
    rem ✘ every piece reads, adds to and writes the same place `index counter 0`
    set (index counter 0) (wrap_add (index counter 0) 1) .
    set i (add i 1) .
  end
  return .
end
```

Output

```
$ lowentc --check mistake_sharedwrite.low
mistake_sharedwrite.low:10:0 E-PAR-WRITE: a splittable loop may only write its OWN eleme
nt `index <s> <i>` — this write can collide with another iteration (Bernstein: wr ∩ wr =
 ∅)
```

Every piece reads `index counter 0`, adds 1 and writes it back. When two threads read the same value and each writes its sum, one increment is lost. That is a write to a place outside the piece’s own share, so it is rejected with `E-PAR-WRITE`. If a shared place really must be updated together, take `cap atomic` and use `atomic_add counter 0 1` (this chapter’s `counter.low`). Usually, though, gathering per-piece counts with `reduce` is faster.

Counter-example. Splitting a sum of floats

examples/ch27/mistake_floatreduce.low

```
module mistake_floatreduce .
rem expect: E-PAR-FLOAT

fn mean input s slice f64 . output f64 .
  parallel s split .
  rem ✘ the result of float addition depends on how it is grouped
  reduce acc add .
```

```
do
  var acc f64 be 0.0 .
  var i u64 be 0 .
  while lt i (len s) . do
    set acc (add acc (index s i)) .
    set i (add i 1) .
  end
  return acc .
end
```

Output

```
$ lowentc --check mistake_floatreduce.low
mistake_floatreduce.low:9:0 E-PAR-IDENTITY: the accumulator this reduction starts from i
s not the IDENTITY of its operator, so a split answer is not the sequential one: every p
iece starts again from that value and it is counted once per piece (measured: sequential
 121, native split 621). DET-1 promises a split is bit-identical — that holds only from
the identity (`add`/`bit_or`/`bit_xor` → 0, `mul` → 1, `max` → 0 on an unsigned width).
Start from the identity and add the offset once, after the loop
mistake_floatreduce.low:9:0 E-PAR-FLOAT: a FLOAT reduction cannot be split: float additi
on is not associative, so the tree shape changes the result and the answer would depend
on the schedule (★ nonassoc_shape_matters, Qed). Use the sequential `sum` (Neumaier) — d
eterminism is part of the meaning, not a detail
mistake_floatreduce.low:4:0 W-PAR-OK: this loop satisfies the Bernstein conditions and m
ay be split (DET-1 proves the parallel result is bit-identical to the sequential one — d
ocs/proofs/coq/LowentPar.v). Execution is still sequential, which is a VALID implementat
ion precisely because of that theorem
```

Floating-point addition is not associative. How the pieces are split changes the rounding, so the answer would depend on the number of cores. Hence `E-PAR-FLOAT`. As the diagnostic suggests, use the sequential `sum_neumaier` (a compensated sum). Determinism is part of the meaning, not a performance option.

Counter-example. Starting a `reduce` accumulator at a value that is not the identity

examples/ch27/mistake_reduceinit.low

```
module mistake_reduceinit .
rem expect: E-PAR-IDENTITY

fn total input s slice u8 . output u64 .
  parallel s split .
  reduce acc add .
do
  rem ✘ the accumulator starts at 100 --- once split, every piece starts at 100
  var acc u64 be 100 .
  var i u64 be 0 .
  while lt i (len s) . do
    set acc (add acc (widen u64 (index s i))) .
    set i (add i 1) .
  end
  return acc .
end
```

Output

```
$ lowentc --check mistake_reduceinit.low
mistake_reduceinit.low:9:0 E-PAR-IDENTITY: the accumulator this reduction starts from is
 not the IDENTITY of its operator, so a split answer is not the sequential one: every pi
ece starts again from that value and it is counted once per piece (measured: sequential
121, native split 621). DET-1 promises a split is bit-identical — that holds only from t
he identity (`add`/`bit_or`/`bit_xor` → 0, `mul` → 1, `max` → 0 on an unsigned width). S
tart from the identity and add the offset once, after the loop
mistake_reduceinit.low:4:0 W-PAR-OK: this loop satisfies the Bernstein conditions and ma
y be split (DET-1 proves the parallel result is bit-identical to the sequential one — do
cs/proofs/coq/LowentPar.v). Execution is still sequential, which is a VALID implementati
on precisely because of that theorem
```

Run sequentially, `total [1,2,3,4,5,6]` is 100 + 21 = 121. Split, **each piece** starts `acc` at 100, and six pieces of native code gave 621 — the promise that a split answer equals the sequential one breaks there. So it is refused with `E-PAR-IDENTITY` (until 2026-09-16 it passed with `W-PAR-OK`). The starting value of a `reduce` must be the **identity** of the gathering operation — 0 for `add`, `bit_or` and `bit_xor`, 1 for `mul`, 0 for `max` on an unsigned width. If there is a value to add, add it to the result outside the loop.

Counter-example. Writing the loop to split in a different shape

examples/ch27/mistake_noloop.low

```
module mistake_noloop .
rem expect: E-PAR-NOLOOP

proc smooth input s mut slice u8 . output u64 . effects none .
  parallel s split .
do
  var i u64 be 0 .
  rem ✘ a loop to split must have the shape `while lt i (len s)` --- and this one reads
a neighbouring element too
  while lt (add i 1) (len s) . do
    set (index s i) (div (wrap_add (index s i) (index s (add i 1))) 2) .
    set i (add i 1) .
  end
  return len s .
end
```

Output

```
$ lowentc --check mistake_noloop.low
mistake_noloop.low:4:0 E-PAR-NOLOOP: the `parallel` clause names a slice, but no `while
lt <i> (len <slice>) . do … end` loop was found to split. That clause is a CLAIM — DET-1
 proves a split is bit-identical only when there IS a loop to split — so with no loop th
e compiler verifies NOTHING while the annotation still tells every reader it was checked
. An unchecked promise is the lie PRINCIPLES.md §0 is about; the same judgement already
refused `mailbox unbounded` and `bounded 0`. Delete the clause (it means nothing here) o
r write the loop it describes
```

The processor recognises only loops of the shape `while lt i (len s) . do … end` as candidates for splitting. `while lt (add i 1) (len s)` is not that shape, so this is `E-PAR-NOLOOP`. As the diagnostic

says, the `parallel` clause is a **claim**, and with no loop to split, nothing is verified and only the claim remains. This loop also reads the neighbouring element `index s (add i 1)`; even with the shape fixed it would be rejected with `E-PAR-READ`. Write neighbour-reading computations (smoothing and the like) as a sequential loop that writes its results into another slice.

Counter-example. Declaring `write_only` and then reading

examples/ch27/mistake_access.low

```
module mistake_access .
rem expect: E-ACCESS-MODE

rem ✘ declared write_only yet it reads; the place may not be filled yet
proc bad_fill input out mut slice u8 . . output u64 . effects state .
  access out write_only .
do
  let x u8 be index out 0 .
  return widen u64 x .
end
```

Output

```
$ lowentc --check mistake_access.low
mistake_access.low:5:0 E-ACCESS-MODE: `access <p> write_only` says this op never READS t
hat place — but it does. A write_only place may be uninitialised: reading it is reading
garbage, and the declaration is what told the caller it was safe
mistake_access.low:7:1 W-EFFECT-OVER: this op DECLARES `state` but never PERFORMS it. A
declared effect is a cost the caller must budget for (a pure caller cannot call an `io`
op). Drop it, or — if a future version will perform it — say so (RFC-0007)
```

A `write_only` place may not be filled yet, so reading it reads garbage. Callers trust the declaration and pass an empty buffer. So it is refused with `E-ACCESS-MODE`. The `W-EFFECT-OVER` that comes along is due to a defect where writes to a caller's buffer are counted as effects inconsistently (chapter 24). If the op must read, drop the mode or take the place as an ordinary input.

A common misconception. Only addition can be gathered with `reduce`

examples/ch27/max_gather.low

```
module max_gather .
rem run: biggest [3,9,2,7]
rem `max` is associative too, with 0 as its identity, so it can be split and gathered
fn biggest input s slice u8 . output u64 .
  parallel s split .
  reduce best max .
do
  var best u64 be 0 .
  var i u64 be 0 .
  while lt i (len s) . do
    set best (max best (widen u64 (index s i))) .
    set i (add i 1) .
  end
  return best .
end
```

Output

```
$ lowentc --run biggest max_gather.low [3,9,2,7]
biggest([3,9,2,7]) = 9
  arg0 (written) = [3,9,2,7]
```

Any associative operation can be gathered. `max` is associative, and 0 is its identity for `u64`, so taking the largest value in each piece and combining them again with `max` gives the same answer. `min`, `mul` and the bit operations work on the same principle. What is rejected is an operation such as `sub`, where grouping changes the answer (`E-PAR-ASSOC`), and floating-point operations (`E-PAR-FLOAT`).

27.9 This chapter's syntax at a glance

Shape	Meaning	Why
`parallel s split .` (op head)	declares that `s` may be split and processed by many	a claim that is checked, not trusted — `W-PAR-OK` when it holds
`while lt i (len s) . do … end`	the shape of a splittable loop	any other shape is `E-PAR-NOLOOP`
reading and writing only `index s i`	only its own share	others' places: `E-PAR-READ` · `E-PAR-WRITE`
`reduce acc add .`	accumulate per piece, then combine with the operation	start at the identity — the operation must be associative (`E-PAR-ASSOC`)
`atomic_add counter 0 1` · `atomic_load cells 0`	atomically on a place named by slice and index	`effects atomic` + `cap atomic`
`… order seq_cst` · `acq_rel` · `acquire` · `release` · `relaxed`	memory ordering — `seq_cst` if unwritten	the easiest to reason about is the default
`order release` on a read, and so on	rejected (`E-ATOMIC-ORDER`)	meaningless combinations are not left undefined
`view_array u64 bytes`	see bytes as a `u64` slice without copying	atomic cells live on an allocated window
`var v vec u32 4 be load xs 0 .` · `reduce_add v`	read four lanes as one value · gather lanes	SIMD within one flow — the lane count is part of the type
`access data shared_read .` · `access out write_only .`	a promise to only read · only write — checked against the body	read-only lets several tasks hold it · breaking it is `E-ACCESS-MODE`

Table 27.4 — Parallel and atomic syntax — shape · meaning · why it looks this way

Recap

`parallel <slice> split .` declares that a loop may be split, and the processor checks that it reads and writes only its own share and does not write places that live across steps. Accumu-

> lation is stated with `reduce <place> <op> .`, and the operation must be associative. The split answer is bit-for-bit identical to the sequential one. When a shared place is updated together, receive `cap atomic` and use atomic operations, choosing a memory ordering with `order`; the default is `seq_cst`, and meaningless combinations are rejected.

Part VIII — The outside world

28 Input, output and files

In this chapter

28.1 Standard output 365
28.2 Reading a whole file 365
28.3 What is opened is closed 366
28.4 End and failure are different answers 368
28.5 Triggering failure on purpose 368
28.6 Common mistakes 369
28.7 This chapter's syntax at a glance 373

What to know first

chapter 16, Capabilities · `cap io`, `cap file_system` and `effects io`
chapter 19, Ownership · values needing completion are finished explicitly
chapter 17, Designing failure · at the boundary, report with `result`

Looking back

In chapter 19, what decided that a type "needs completion"?

> **A.** The existence of an op that takes the type as `owned` and returns a `result`. Letting such a value go out of scope without finishing it was rejected with `E-OWN-INCOMPLETE`. This chapter shows where that rule is most useful — files that must be closed once opened.

The need for this chapter, and its context

Part VIII is about the places a program meets the outside world, and the first is input/output. It is where failure is common — files are missing, disks fill up, reads get cut off. And the most common defects are mixing failure with **the end** and forgetting to close. Lowent uses capabilities to decide who may do input/output, ownership to make closing unforgettable, and the shape of answers to separate end from failure. Tools built up over the previous four parts come together here.

By the end of this chapter

You will learn to write to standard output with `write_out`, and to open, close, read and write files with the standard library's `files`. You will confirm that a file handle is an owned value needing completion, so forgetting to close it is refused at translation. You will also see why a read's answer splits three ways, "read · end · failure", and how to trigger failures on purpose to test them.

The questions this chapter answers

1 How do you read a file larger than the buffer?

28.1 Standard output

The only channel to standard output is `write_out <cap io> <fd> <bytes>`. File descriptor 1 is standard output and 2 is standard error. It returns the number of bytes written. It has already appeared in several chapters, so there is nothing new, but one point is worth making. That this op takes the capability as its **first operand** means `cap io` shows in the head of every op that emits even one byte to standard output. There is no debug print slipped in secretly.

28.2 Reading a whole file

examples/ch28/lines.low

```
module lines .
rem run: main

use files .

fn is_newline input c u8 . output bool .
do
  return eq c 10 .
end

proc main input out cap io . input fs cap file_system . input al cap allocator .
  output u8 .
  effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 4096 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  let r result u64 files.file_error be files.slurp fs "notes.txt" buf .
  guard not (is_error r) . else do
    let e u64 be write_out out 1 "cannot read notes.txt\n" .
    return 2 .
  end
  let body slice u8 be subslice buf 0 (ok_value r) .
  let n u64 be pipe body do
    filter is_newline .
    count .
  end .
  let w u64 be write_out out 1 body .
  return narrow u8 n .
end
```

Output

```
$ lowentc --run main lines.low
first line
second line
third line
main() = 3
```

- `main` receives three capabilities: output (`cap io`), files (`cap file_system`) and allocation for the buffer (`cap allocator`). From the head alone you know this program does not touch the network.

- `files.slurp fs "notes.txt" buf` reads the whole file into `buf` and gives the number of bytes read as a `result u64 files.file_error`. If the file is missing or larger than the buffer, it is an error. It does not truncate.
- On error it writes a message and returns exit code 2 — the shape of handling failure at the boundary (chapter 17).
- It cuts off the part read with `subslice`, counts line breaks with `pipe` (chapter 24), and writes the contents to standard output as is.

The example reads `notes.txt` from its own folder. It has three lines, so the exit code is 3.

> **Q. How do you read a file larger than the buffer?**
>
> **A.** Open it with `files.open` and repeat `files.read`. `read` reads only as much as the buffer, so a file of any size can be read in pieces. A short read is not a failure; giving less than asked is normal, and the caller must keep reading. `slurp` does exactly that inside.

Reading the 34-byte `notes.txt` both ways gives this (measured on this edition with different buffer sizes).

```
files.slurp  buf 64  ──►  ok 34
files.slurp  buf 34  ──►  ok 34
files.slurp  buf 33  ──►  error buffer_too_small
files.read   buf 16  ──►  ok (some 16) · ok (some 16) · ok (some 2) · ok none
```

`slurp` answers `buffer_too_small` when the buffer is even one byte smaller than the file; an exact fit is fine. `read` hands out pieces whatever the buffer size, and joining them is the caller's job.

28.3 What is opened is closed

examples/ch28/copy.low

```
module copy .
rem run: main

use files .

proc main input out cap io . input fs cap file_system . output u8 . effects io .
do
  let o result files.handle files.file_error be files.open fs "../../build/ch28-copy.txt
" 1 .
  guard is_ok o . else return 1 .
  var h owned files.handle be ok_value o .
  let w result u64 files.file_error be files.write fs h "hello, file\n" .
  let c result void files.file_error be files.close fs h .
  guard is_ok w . else return 2 .
  guard is_ok c . else return 3 .
  let m u64 be write_out out 1 "wrote and closed\n" .
  return 0 .
end
```

Output

```
$ lowentc --run main copy.low
wrote and closed
main() = 0
```

- `files.open fs <path> 1` opens for writing. Modes are 0 read, 1 write (truncate and create), 2 append.
- On success, `ok_value o` is put into an `owned files.handle`.
- It writes with `files.write` and closes with `files.close fs h`. `close` takes an `owned handle` and returns a `result`. Closing really can fail — on network file systems or full disks, failing to flush the last buffer shows up at close.

Because `close` has that shape, `handle` is a type needing completion. Leaving it unclosed is rejected.

examples/ch28/forgot.low

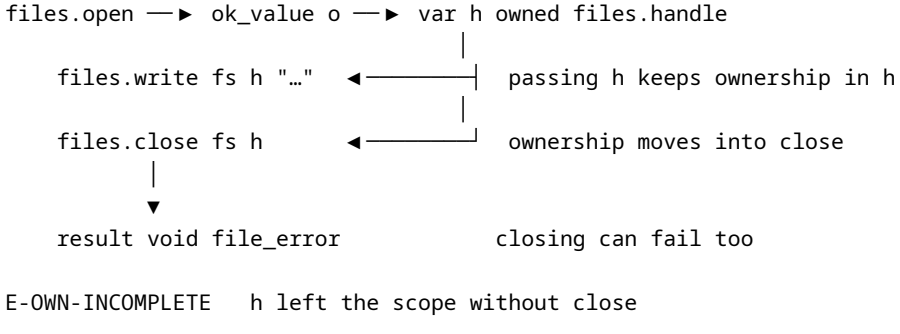
```
module forgot .
rem expect: E-OWN-INCOMPLETE

use files .

proc leak input fs cap file_system . output u8 . effects io .
do
  let o result files.handle files.file_error be files.open fs "notes.txt" 0 .
  guard is_ok o . else return 1 .
  var h owned files.handle be ok_value o .
  return 0 .
end
```

Output

```
$ lowentc --check forgot.low
10:0 E-OWN-INCOMPLETE: this value is dropped automatically at the end of scope — but the
 program itself declares an op that takes this type `owned` BY VALUE and returns a `resu
lt`: finishing it CAN FAIL. An automatic drop is a RELEASE (total, non-suspending), and
it has nowhere to hand you that failure — it would SWALLOW it. A fallible finish (flush/
commit/close) is a COMPLETION and must be EXPLICIT: call it and handle the `result`. If
you really mean to discard the value and its failure, say so with `drop`. RFC-0058
```

A program that forgets to close does not translate. It does not count on the operating system closing file descriptors when the process ends. If you really mean to discard without closing, write `drop h .` to say so.

The life of one handle, on one page:



```
E-OWN-MOVED        h was used again after close
drop h .           discard it on purpose, unclosed
```

Translation rejects the first two. The third is how the intent to discard is left in the source.

> **A common misconception. Carrying a file handle as a plain integer is lighter**
>
> Holding the integer the kernel gave has the same run-time cost; `files.handle` is a struct holding one integer. The difference is what translation can know. An integer can be copied, forgotten or closed twice without anyone noticing. An owned value moves, is rejected when forgotten, and is rejected when closed twice. Rules are put onto the same bytes.

28.4 End and failure are different answers

`files.read` answers in three positions.

Answer	Meaning
`ok (some n)`	`n` bytes were read
`ok none`	The **end** of the file. Not a failure
`error e`	A failure. `e` says which operation failed

Table 28.1 — The answers of `read`

Reading a 34-byte file again and again with a 16-byte buffer gives these answers (measured on this edition).

```
notes.txt  34 bytes            buf  16 bytes

read 1   ok (some 16)    bytes  0 .. 15
read 2   ok (some 16)    bytes 16 .. 31
read 3   ok (some 2)     bytes 32 .. 33
read 4   ok none         the end — stop here
read 5   ok none         reading past the end is still the end
```

A last piece (2 bytes) smaller than the buffer does not announce the end. The end is announced separately, by `ok none`.

At one time this module's `slurp` just stopped its loop when a read failed. A failed read was then reported as "read the whole file", and a line-counting program built on the module reported a read failure as the success "0 lines". The lesson was that one value must not carry two meanings (end and failure), and so the answer became three positions.

Nor were three positions mechanically put on every op. Opening has no "end", so `open` is two positions, `result handle file_error`. The reader in the `io` module, which reads slices in memory, has nowhere to fail, so it uses only `option`. The honest shape differs per op.

28.5 Triggering failure on purpose

File failures rarely happen normally, so code that handles them tends to stay untested. The standard library's host operations have a fault injector, switched on with an environment variable.

```
LOW_HOST_FAULT="open:err"        every open fails
LOW_HOST_FAULT="read:err@2"      the second read fails
LOW_HOST_FAULT="read:short@1=4"  the first read is cut to 4 bytes
LOW_HOST_FAULT="close:err"       closing fails
```

The VM and native code use the same injector, so the two back ends must agree. Running `LOW_HOST_FAULT="open:err" lowentc --run main lines.low` sends `lines.low` down its error branch, writes the message and returns 2. It is off by default.

The 16-byte reads from the previous section, run again under the injector, change like this (measured on this edition).

```
(no fault)          ok 16 · ok 16 · ok 2  · ok none
read:short@1=4      ok 4  · ok 16 · ok 14 · ok none
read:err@2          ok 16 · error read_failed
```

A short read still adds up to 34, and the end is still announced by `ok none`. `eof_fixed.low` returns 34 on both the first and the second row. On the third row the failure arrives as an answer different from the end, so only code that tells the two apart takes this path correctly.

> **In practice. A defect found by a line-counting program**
>
> The injector made the “0 lines” defect above visible. Failing the first read with `LOW_HOST_FAULT="read:err@1"` made the line count finish successfully. Had the failure path never run, this defect would have stayed hidden until a real disk error. If capabilities and ownership stop “forgetting to close” at translation, the injector reveals “handling failure wrongly” at run time.

28.6 Common mistakes

> **Counter-example. Opening in read mode and writing — without checking the byte count**

examples/ch28/mistake_readmode.low

```
module mistake_readmode .
rem run: main

use files .

proc main input out cap io . input fs cap file_system . output u8 . effects io .
do
  rem ✘ mode 0 is read --- yet it writes
  let o result files.handle files.file_error be files.open fs "notes.txt" 0 .
  guard is_ok o . else return 1 .
  var h owned files.handle be ok_value o .
  let w result u64 files.file_error be files.write fs h "extra line\n" .
  let c result void files.file_error be files.close fs h .
  drop c .
  guard is_ok w . else return 2 .
  rem the write failed, so it leaves with 2 here --- a short write is checked too
  guard eq (ok_value w) 11 . else return 3 .
  return 0 .
end
```

Output

```
$ lowentc --run main mistake_readmode.low
main() = 2
```

`files.open fs "notes.txt" 0` is read mode. Writing there makes `files.write` return a failure — a short write with the stream's error flag set is a failure, not a value — and this example leaves with exit code 2. The same spot used to answer `ok 0` ("0 bytes written"), so code that asked only `is_ok w` passed it as a success. Check the mode (0 read · 1 write · 2 append), and look at **the number of bytes written** as well as success. A short write means the remaining bytes must be written again.

Counter-example. Writing with a handle that was already closed

examples/ch28/mistake_afterclose.low

```
module mistake_afterclose .
rem expect: E-OWN-MOVED

use files .

proc main input out cap io . input fs cap file_system . output u8 . effects io .
do
  let o result files.handle files.file_error be files.open fs "../../build/ch28-after.tx
t" 1 .
  guard is_ok o . else return 1 .
  var h owned files.handle be ok_value o .
  let c result void files.file_error be files.close fs h .
  drop c .
  rem ✘ writes again with the closed handle --- `close` took ownership
  let w result u64 files.file_error be files.write fs h "late\n" .
  drop w .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_afterclose.low
mistake_afterclose.low:14:0 E-OWN-MOVED: this `owned` value was already MOVED (consumed)
 — using it again is use-after-move, which SPEC-004 §4.8 has always called a compile err
or and which nothing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set <n
ame> <new value>` re-initialises the place — that is how a handle threads through a loop
), or borrow it LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered y
et (E-IR-UNSUP says so at the call site), so `f (ref h)` is not a way out today
```

`files.close` takes an `owned handle`, so ownership passes the moment it is called. Afterwards `h` is a handle that no longer exists, so this is `E-OWN-MOVED`. In C, `fwrite` after `fclose` is undefined behaviour, and if the same number was reused by a newly opened file, the bytes land in the wrong file. Ownership stops that at translation time.

Counter-example. Asking only about failure in the answer of a read

examples/ch28/mistake_eof.low

```
module mistake_eof .
rem trap: main

use files .

proc main input out cap io . input fs cap file_system . input al cap allocator .
  output u8 .
  effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 64 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  let o result files.handle files.file_error be files.open fs "notes.txt" 0 .
  guard is_ok o . else return 2 .
  var h owned files.handle be ok_value o .
  var total u64 be 0 .
  var i u64 be 0 .
  while lt i 3 . do
    let r result (option u64) files.file_error be files.read fs h buf .
    guard is_ok r . else return 3 .
    rem ✘ only failure is checked --- `some_value` stops at the end-of-file `ok none`
    set total (add total (some_value (ok_value r))) .
    set i (add i 1) .
  end
  let c result void files.file_error be files.close fs h .
  drop c .
  return narrow u8 total .
end
```

Output

```
$ lowentc --run main mistake_eof.low
== ir diagnostics (1) ==
0:0 E-VM-NONE: some_value of none (panic)
```

The answer of `files.read` has three places. `is_ok r` only says “not a failure”. The end-of-file `ok none` is not a failure either, so it passes, and taking `some_value` out of it stops the program. Split all three places.

examples/ch28/eof_fixed.low

```
module eof_fixed .
rem run: main

use files .

proc main input out cap io . input fs cap file_system . input al cap allocator .
  output u8 .
  effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 16 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  let o result files.handle files.file_error be files.open fs "notes.txt" 0 .
  guard is_ok o . else return 2 .
```

```
  var h owned files.handle be ok_value o .
  var total u64 be 0 .
  var more bool be true .
  var rounds u64 be 0 .
  while and more (lt rounds 100) . do
    let r result (option u64) files.file_error be files.read fs h buf .
    guard is_ok r . else do
      let c1 result void files.file_error be files.close fs h .
      drop c1 .
      return 3 .
    end
    rem split all three places --- read · end · failure
    let got option u64 be ok_value r .
    if is_some got . do
      set total (add total (some_value got)) .
    end else do
      set more false .
    end
    set rounds (add rounds 1) .
  end
  let c result void files.file_error be files.close fs h .
  guard is_ok c . else return 4 .
  return narrow u8 total .
end
```

Output

```
$ lowentc --run main eof_fixed.low
main() = 34
```

It reads the 34-byte file in three pieces with a 16-byte buffer and stops at the `ok none` of the fourth read. On failure it closes and returns 3; if closing fails, it returns 4. The bound on `rounds` makes the loop end even for a source that never reports its end.

A common misconception. `slurp` of a file larger than the buffer reads just the beginning

examples/ch28/slurp_small.low

```
module slurp_small .
rem run: main

use files .

proc main input out cap io . input fs cap file_system . input al cap allocator .
  output u8 .
  effects io alloc .
do
  rem an 8-byte buffer for a 34-byte file
  let g option mut slice u8 be alloc_bytes al capacity 8 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  let r result u64 files.file_error be files.slurp fs "notes.txt" buf .
  rem it does not come back cut to the first 8 bytes --- it is an error
  guard not (is_error r) . else return 2 .
```

```
  return narrow u8 (ok_value r) .
end
```

Output

```
$ lowentc --run main slurp_small.low
main() = 2
```

`slurp` of a 34-byte file into an 8-byte buffer returns an error, not the first 8 bytes. This prevents the bug of trusting truncated content as the whole file — the tail of a configuration file silently disappearing. If a file may be larger than the buffer, read it in pieces with `files.open` and `files.read`.

28.7 This chapter's syntax at a glance

Shape	Meaning	Why
`write_out out 1 "…"`	write to standard output (1) · standard error (2) — returns the count	the capability comes first — no hidden printing
`use files . + input fs cap file_system .`	the file module and its capability	the head shows whether files are reached
`files.open fs "notes.txt" 0`	open — 0 read · 1 write · 2 append · `result handle file_error`	opening has no "end" — two places
`var h owned files.handle be ok_value o .`	hold the handle as owned	forgetting it: `E-OWN-INCOMPLETE` · writing after close: `E-OWN-MOVED`
`files.read fs h buf`	`ok (some n)` read · `ok none` end · `error e` failure	one value never carries two meanings
`files.write fs h bytes`	returns the count as a `result`	writes can be short — check the count
`files.close fs h`	takes `owned`, returns `result` — completion	closing can fail too
`files.slurp fs path buf`	read it whole — an error if larger than the buffer	never truncates
`LOW_HOST_FAULT="read:err@2"`	inject failures on purpose (environment variable)	exercise the failure paths

Table 28.2 — I/O and file syntax — shape · meaning · why it looks this way

Recap

Standard output is `write_out <cap io> <fd> <bytes>`, with the capability as first operand. The `files` module takes `cap file_system` to open, close, read and write files. A file handle needs completion because `close` takes it `owned` and returns a `result`, so forgetting to close is refused

at translation. `read` answers in three positions — read, end, failure — and failure paths are exercised on purpose with the `LOW_HOST_FAULT` injector.

29 Meeting C

In this chapter

29.1 Calling C — mark, right, effects line 375
29.2 Types that cross the boundary 377
29.3 C calls Lowent 377
29.4 Callbacks and ownership 379
29.5 Common mistakes 380
29.6 This chapter's syntax at a glance 383

What to know first

chapter 16, Capabilities · `cap c` is a capability the entry point cannot receive
chapter 14, Contracts · `requires` is checked on entry
chapter 21, Modules · only `export`ed ops become symbols C can call

Looking back

Why did chapter 16 say `cap c` is a capability the entry point cannot ask for?

> **A.** Because it is not a capability the runner (the operating system) can hand over. The door into C must be made somewhere entitled to give it and flow in as an argument. This chapter covers that door — `extern` — in both directions, going out and coming in.

The need for this chapter, and its context

For a new language to be used in practice, it must meet existing C code. Operating system APIs, decades-old libraries and hardware vendors' SDKs are all C. But the moment control passes into C, everything this language kept — bounds checks, ownership, effects — can break on the C side. Lowent does not make this place comfortable; it makes it **narrow and visible**. And in the other direction, where C calls Lowent, it puts contracts at the door to guard the inside.

By the end of this chapter

You will learn the three things an `extern` op calling a C function must have (the `unsafe` mark, `cap c` and an effects line) and the `link` clause. You will see that types crossing the boundary are limited to what the C ABI can express, and that a slice becomes a pointer and a length. You will put `export`ed ops into a C program with `--emit-h` and `--no-main` and watch a C caller that breaks a contract stop at the door. You will also see the rules for callbacks and handing over ownership.

The questions this chapter answers

1 Inside an op marked `unsafe`, may you do anything?

29.1 Calling C — mark, right, effects line

examples/ch29/area.low

```
module area_ffi .

rem the body lives in C — it carries all three: marker, rights and effects
unsafe extern proc c_area do
  input k cap c .
  input w i64 .
  input h i64 .
  output i64 .
  effects unsafe .
  link "lw_c_area" .
end

unsafe proc area_twice input k cap c . input w i64 . input h i64 . output i64 .
  effects unsafe .
do
  return add (c_area k w h) (c_area k w h) .
end
```

Output

```
$ lowentc --check area.low
== check: ok ==
```

An op that calls a C function has **all** three.

Must have	Without it	Whom it tells
the `unsafe` mark	`E-FFI-NOUNSAFE`	The person — from here on a person, not the language, is responsible
`input k cap c .`	`E-FFI-NOCAP`	The processor — entering C is a right handed over
an effects line (at least `effects unsafe`)	`E-FFI-NOEFFECT`	The caller — it learns from the head what it takes on

Table 29.1 — What an op calling C must have

An `extern` op's body is in C, so there is no body here. Instead, the way a `struct` holds its fields, it holds its **clauses** in `do … end`, and among them `link "lw_c_area"` names the C side. A statement in that block would be a second body: `E-FFI-BODY`. The processor does not make the name up from the op's name. It is a promise made to another language, so whoever promises writes it — leaving it out is `E-FFI-LINK`. `area_twice` calls that op, so it is itself `unsafe` and receives `cap c`. The mark and the right travel up the call chain.

examples/ch29/nounsafe.low

```
module nounsafe .
rem expect: E-FFI-NOUNSAFE

extern proc c_area do
  input k cap c .
  input w i64 .
  input h i64 .
  output i64 .
  effects unsafe .
  link "lw_c_area" .
end
```

Output

```
$ lowentc --check nounsafe.low
nounsafe.low:4:0 E-UNSAFE-UNDECLARED: this op declares the `unsafe` EFFECT but is not ma
rked `unsafe`. The effect says WHAT it does; the modifier says WHO takes responsibility.
 Write `unsafe proc …` — an unsafe op that nobody signed for is exactly the hole the dis
cipline exists to close
nounsafe.low:4:0 E-FFI-NOUNSAFE: calling C without `unsafe`. Inside that C function ever
y invariant this language enforces can be broken, and the tool cannot see it. What canno
t be checked must at least be MARKED (RFC-0063 D1)
```

examples/ch29/nocap.low

```
module nocap_ffi .
rem expect: E-FFI-NOCAP

unsafe extern proc c_area do
  input w i64 .
  input h i64 .
  output i64 .
  effects unsafe .
  link "lw_c_area" .
end
```

Output

```
$ lowentc --check nocap.low
nocap.low:4:0 E-FFI-NOCAP: this op calls C but receives NO right to do so. Calling into
C is a capability you are HANDED — `input k cap c .` — exactly like the heap (RFC-0043),
 the device bus (RFC-0042) and raw machine instructions (RFC-0041). An ambient door into
 C is one every caller silently inherits
```

With only one of the three, the call “could work”. But then this language could not say what it has given up.

> **Q. Inside an op marked `unsafe`, may you do anything?**
>
> **A.** No. `unsafe` does not mean “anything goes”; it marks that “part of this place cannot be checked by the processor”. In the body of an `unsafe` op, bounds checks, ownership and borrowing rules still apply. What cannot be checked is **inside** the C function. The mark stays in the source, so the places to audit can be found later.

29.2 Types that cross the boundary

Only what the C ABI can express crosses the boundary. `option`, `result`, vectors and containers do not exist in C, and the processor does not pretend they do.

examples/ch29/fftype.low

```
module fftype .
rem expect: E-FFI-TYPE

unsafe extern proc c_find do
  input k cap c .
  input key u64 .
  output option u64 .
  effects unsafe .
  link "lw_c_find" .
end
```

Output

```
$ lowentc --check fftype.low
fftype.low:4:0 E-FFI-TYPE: this signature carries a type the C ABI cannot express (optio
n / result / vector / container). C has no such thing, and the tool will not PRETEND it
does: pass integers, f64, or a byte slice (which becomes a POINTER and a LENGTH — two C
arguments, because that is what a slice honestly IS in C)
```

Only slices are mapped automatically. `slice t` becomes two arguments, "a pointer to `t`" and "a count". The width is known too — `slice u32` is `uint32_t *`, not a byte pointer. Capabilities are not values, so they do not cross into C. There is one ABI name, `c`. "What C is on this machine" is already decided by the machine building it.

29.3 C calls Lowent

The other direction. An `export`ed op becomes a symbol callable from C.

examples/ch29/exported.low

```
module exported .
rem run: clamp_add 100 50

export fn clamp_add input a u32 . input b u32 . output u32 .
  requires le a 1000 .
  requires le b 1000 .
  ensures le ret 2000 .
do
  return add a b .
end

export fn sum_bytes input xs slice u8 . output u64 .
do
  var s u64 be 0 .
  for x xs do
    set s (add s (widen u64 x)) .
  end
```

```
  return s .
end
```

Output

```
$ lowentc --run clamp_add exported.low 100 50
clamp_add(100, 50) = 150
```

`--emit-h` emits the header. A header written by hand keeps signatures in two places, and one day they diverge.

```
long long clamp_add(long long, long long);
long long sum_bytes(const unsigned char *, size_t);
```

You can see `sum_bytes`'s `slice u8` split into a pointer and a length.

```
Lowent
  input xs slice u8 .

C (the header --emit-h writes)
  sum_bytes(const unsigned char *, size_t)
            └──────────┬─────────┘ └──┬──┘
                       │              count (len xs)
                  pointer (xs's first item)
```

`--no-main` emits C holding only the entry points of exported ops, without `main` and the command-line dispatcher, ready to drop into someone else's build. The following C program uses the two.

```
// host-for: exported.low
#include <stdio.h>
#include "lowent.h"

int main(void) {
    const unsigned char bytes[] = {1, 2, 3, 4};
    printf("clamp_add(100, 50) = %lld\n", clamp_add(100, 50));
    printf("sum_bytes = %lld\n", sum_bytes(bytes, 4));
    fflush(stdout);
    printf("clamp_add(5000, 1) = %lld\n", clamp_add(5000, 1));
    return 0;
}
```

```
$ cc host.c <exported.low 을 --no-main 으로 낸 C> && ./host
clamp_add(100, 50) = 150
sum_bytes = 10
panic: requires violated at entry
(종료 코드 70)
```

The first two calls get answers. The third breaks `requires le a 1000 .`. C does not know contracts, but **the place being called into is this side's door**, so the op's contract is enforced on the arguments and it stops on entry. Lowent answers for inside the door, C for outside.

Both directions in one picture. Going out, it is **written in the head**; coming in, it is **measured at the door**.

```
going out --- Lowent calls C
  area_twice ──▶ c_area ══ link "lw_c_area" ══▶  lw_c_area (C)
  unsafe · cap c · effects unsafe               inside is unchecked; a person answers
```

```
coming in --- C calls Lowent
  host.c ── clamp_add(5000, 1) ═══▶ ┃ requires le a 1000 .
                                    ┃ 5000 breaks the promise
                                    ┗━▶ stops at entry (exit code 70)
```

> **A common misconception. At the FFI boundary every guarantee of the language disappears**
>
> Going **out** to C, the inside of the C function is outside the checks. But that fact is written in the head with `unsafe`, `cap c` and an effects line. Where C comes **in**, the guarantees stand as they are. Incoming arguments that break a contract stop on entry. The boundary is not a hole but a door, and a contract stands at the door.

29.4 Callbacks and ownership

To let C call a Lowent function back, take the address of an `export extern` op with `unsafe_fn <op>` and pass it as a value. There is no new convention. Two rules apply.

- `unsafe_fn` may point only at `export extern` ops (`E-FN-NOTEXPORT`). The address of a door C cannot call is not an address.
- An op used as a callback cannot require capabilities (`E-FN-CAP`). It is C that enters the callback, and C has no capabilities to hand over. Work needing capabilities is done outside the callback; inside, it only computes.

Passing `owned T` to an `extern` moves the responsibility to dispose of it to C. This side's obligation ends there, and whether it is later released is not verified. C functions taking an unfixed number of arguments after the fixed ones can be called with a `variadic .` clause, but contracts do not reach those arguments, and the opposite direction (C calling our variadics) does not exist.

examples/ch29/variadic.low

```
module variadic .

rem C's char*; unsafe_ptr is a qualifier placed before a type
newtype cstr unsafe_ptr u8 .

unsafe extern proc c_printf do
  input k cap c .
  input fmt cstr .
  output i32 .
  effects unsafe .
  rem takes an unspecified number of arguments after the fixed ones
  variadic .
  link printf .
end

rem after the fixed argument (the format) one more value, 42, is passed; neither type ch
ecks nor contracts reach it
unsafe proc report input k cap c . output i32 . effects unsafe .
do
  return c_printf k (cstr_of "sum=%d\n\0") 42 .
end
```

Output

```
$ lowentc --check variadic.low
== check: ok ==
```

- `newtype cstr unsafe_ptr u8 .` turns C's `char*` into a named type. `unsafe_ptr` is not a type used on its own but a qualifier placed before a type.
- The `variadic .` clause says "takes more after the fixed arguments", and `link printf .` writes the C name. The `42` at the call site follows the fixed arguments, so neither type checks nor contracts reach it. The classic C defect of a format not matching its arguments is not stopped here, and that is why `unsafe` is attached.
- `cstr_of "…\0"` views a string literal ending in a zero byte as a C string. The VM cannot call C, so this example is only checked; built natively, it prints `sum=42`.

> **In practice. The load carried by a program that adds two numbers**
>
> The default emission is a **program**. `main` and the command-line dispatcher come along, and the dispatcher holds the tagged path and its pools. In the development repository's measurement, emitting one op that adds two numbers by default gave about 197 KB of read-only data and about 1.7 MB of uninitialised data, while `--no-main` brought both under 50 bytes. That is why `--no-main` is used when embedding as a library. Targets without an operating system (`--target cortex_m`) do not emit the dispatcher in the first place.

29.5 Common mistakes

> **Counter-example. Leaving the effects line off an `extern` op**
>
> **examples/ch29/mistake_noeffect.low**
>
> ```
> module mistake_noeffect .
> rem expect: E-FFI-NOEFFECT
>
> rem ✘ no effects line --- callers cannot learn from the head what they take on
> unsafe extern proc c_area do
>   input k cap c .
>   input w i64 .
>   input h i64 .
>   output i64 .
>   link "lw_c_area" .
> end
> ```
>
> **Output**
>
> ```
> $ lowentc --check mistake_noeffect.low
> mistake_noeffect.low:5:0 E-UNSAFE-UNUSED: this op is marked `unsafe` but declares no `un
> safe` effect. An `unsafe` that buys nothing is a FALSE ALARM — and false alarms are how
> real ones stop being read. Drop the modifier
> mistake_noeffect.low:5:0 E-FFI-NOEFFECT: this op calls C and declares no effect. The eff
> ect row is how a CALLER learns what it is taking on; C that hides in a clean signature i
> s the hidden cost this language exists to remove (declare at least `effects unsafe`)
> ```

Two diagnostics appear. `E-FFI-NOEFFECT` says “this calls C and declares no effect”, and `E-UNSAFE-UNUSED` says “this is marked `unsafe` but has no `unsafe` effect, so it is a false alarm”. Both have the same root. The marker (`unsafe`) says **who takes responsibility**, the effects line (`effects unsafe`) says **what it does**, and one without the other does not pair up. A single `effects unsafe .` line removes both.

Counter-example. Leaving the `link` clause off an `extern` op

examples/ch29/mistake_nolink.low

```
module mistake_nolink .
rem expect: E-FFI-LINK

rem ✘ no `link` clause --- nothing says which C symbol is called
unsafe extern proc c_area do
  input k cap c .
  input w i64 .
  input h i64 .
  output i64 .
  effects unsafe .
end

unsafe proc area_twice input k cap c . input w i64 . input h i64 . output i64 .
  effects unsafe .
do
  return add (c_area k w h) (c_area k w h) .
end
```

Output

```
$ lowentc --check mistake_nolink.low
mistake_nolink.low:5:0 E-FFI-LINK: an `extern` op does not say which C symbol it calls.
Write `link "strlen" .` (and `link "sin" from "m" .` when it lives in a library). The C
name is a promise made to another language, so the author writes it: derived from the op
 name it would change the moment the op is renamed, with nothing in the source saying so
 (RFC-0063)
```

Without `link`, nothing in the source says which C symbol is called. If the tool derived it from the op name, renaming the op would call a **different C function** with nothing recording that. So it is rejected with `E-FFI-LINK` — the promise belongs to whoever writes it.

Counter-example. Calling an op that calls C without the `unsafe` marker

examples/ch29/mistake_callerunsafe.low

```
module mistake_callerunsafe .
rem expect: E-UNSAFE-UNDECLARED

unsafe extern proc c_area do
  input k cap c .
  input w i64 .
  input h i64 .
  output i64 .
  effects unsafe .
```

```
  link "lw_c_area" .
end

rem ✘ calls an op that calls C, but leaves out the `unsafe` marker
proc area_twice input k cap c . input w i64 . input h i64 . output i64 .
  effects unsafe .
do
  return add (c_area k w h) (c_area k w h) .
end
```

Output

```
$ lowentc --check mistake_callerunsafe.low
mistake_callerunsafe.low:14:0 E-UNSAFE-UNDECLARED: this op declares the `unsafe` EFFECT
but is not marked `unsafe`. The effect says WHAT it does; the modifier says WHO takes re
sponsibility. Write `unsafe proc …` — an unsafe op that nobody signed for is exactly the
 hole the discipline exists to close
```

`area_twice` declares `effects unsafe` but has no `unsafe` in its head. Effects travel up the calls, so the calling op performs the `unsafe` effect too, and whatever performs it must be signed for. As `E-UNSAFE-UNDECLARED` puts it, “an unsafe op that nobody signed for is exactly the hole” this language exists to close. Write `unsafe proc area_twice …`.

Counter-example. Passing the address of an ordinary op as a callback

examples/ch29/mistake_cbplain.low

```
module mistake_cbplain .
rem expect: E-FN-NOTEXPORT

fn by_value input a i64 . input b i64 . output i64 .
  requires ge a -1000000 .
  requires le a 1000000 .
  requires ge b -1000000 .
  requires le b 1000000 .
do
  return sub a b .
end

unsafe proc callback_addr input k cap c . output u64 . effects unsafe .
do
  rem ✘ tries to hand C the address of an ordinary op --- it has no door C can call
  let f u64 be unsafe_fn by_value .
  return f .
end
```

Output

```
$ lowentc --check mistake_cbplain.low
mistake_cbplain.low:14:1 W-EFFECT-OVER: this op DECLARES `unsafe` but never PERFORMS it.
 A declared effect is a cost the caller must budget for (a pure caller cannot call an `i
o` op). Drop it, or — if a future version will perform it — say so (RFC-0007)
mistake_cbplain.low:16:0 E-FN-NOTEXPORT: `unsafe_fn <op>` needs an `export extern` op —
```

```
only those have a C-callable symbol whose entry checks the contract (RFC-0066 §2). A pla
in op has no address C can call
```

C can call only the symbols made by `export extern` ops, and at that entry the contract is enforced on the arguments. An ordinary op has no such door, hence `E-FN-NOTEXPORT`. Change the head to `export extern fn by_value …`. The `W-EFFECT-OVER` on the first line says that merely taking an address does no `unsafe` work; `unsafe` arises in the op that calls C with that address.

Counter-example. Giving a callback op a capability input

examples/ch29/mistake_cbcap.low

```
module mistake_cbcap .
rem expect: E-FN-CAP

rem the op meant as a callback takes a capability as input
export extern fn cmp_logged input out cap io . input a i64 . input b i64 . output i64 .
  requires ge a -1000000 .
  requires le a 1000000 .
  requires ge b -1000000 .
  requires le b 1000000 .
do
  return sub a b .
end

unsafe proc callback_addr input k cap c . output u64 . effects unsafe .
do
  rem ✘ when C calls back, it has no capability to hand over
  let f u64 be unsafe_fn cmp_logged .
  return f .
end
```

Output

```
$ lowentc --check mistake_cbcap.low
mistake_cbcap.low:15:1 W-EFFECT-OVER: this op DECLARES `unsafe` but never PERFORMS it. A
 declared effect is a cost the caller must budget for (a pure caller cannot call an `io`
 op). Drop it, or — if a future version will perform it — say so (RFC-0007)
mistake_cbcap.low:17:0 E-FN-CAP: `unsafe_fn <op>` names an op that requires a CAPABILITY
 (a `cap` parameter) — but a callback is entered BY C, and C has no capability to hand i
t. Do the capability-needing work OUTSIDE the callback and keep the callback pure or lim
ited-effect (RFC-0066)
```

It is C that enters a callback, and C has no capability to hand over. So an op that takes a capability cannot be a callback (`E-FN-CAP`). Do capability-needing work, such as output or files, **outside** the callback, after C returns. Inside the callback, only compute.

29.6 This chapter's syntax at a glance

Shape	Meaning	Why

`unsafe extern proc c_area do input k cap c . … effects unsafe . link "lw_c_area" . end`	an op whose body is in C — its clauses go in `do … end`	marker, right and effects line must all be present
`unsafe proc area_twice input k cap c . … effects unsafe .`	an op that calls an op that calls C	marker and right travel up the call chain
`input xs slice u8 .` **(at the boundary)**	two arguments in C: pointer and length	only slices are mapped automatically
`option· result·` **vectors at the boundary**	rejected (`E-FFI-TYPE`)	nothing absent from the C ABI is faked
`export fn clamp_add …`	a symbol C can call	the contract is enforced on incoming arguments
`lowentc --emit-h · --no-main`	emit a header · emit as a library without `main`	the signature lives in one place
`unsafe_fn cmp`	address of an `export extern` op (callback)	ordinary op: `E-FN-NOTEXPORT` · with a capability: `E-FN-CAP`
`input h owned t .` **(to an extern)**	responsibility for destroying passes to C	what C does with it afterwards is not verified
`variadic .`	call a C variadic function	contracts do not reach variadic arguments
`newtype cstr unsafe_ptr u8 . · cstr_of "…\0"`	C pointer qualifier · view as a zero-terminated C string	pointers are handled only through named types

Table 29.2 — C boundary syntax — shape · meaning · why it looks this way

Recap

An `extern` op calling C has all of `unsafe`, `cap c` and an effects line, and names the C symbol with `link`. Only what the C ABI can express crosses the boundary, and a slice becomes a pointer and a length. `export`ed ops go into C with `--emit-h` and `--no-main`, and a C caller breaking a contract stops at the door. Callbacks use the address of an `export extern` op without capabilities, and passing `owned` hands responsibility to C.

30 Hardware — registers, interrupts, machine instructions

In this chapter

30.1 The register map 385
30.2 Access is enforced at translation 387
30.3 Interrupt handlers 388
30.4 What a machine supports — `build tier` 389
30.5 Machine instructions — `asm` 390
30.6 The absorbing boundary — where `unsafe` stops 392
30.7 Using what the machine has — `--hw` 393
30.8 Common mistakes 393
30.9 This chapter's syntax at a glance 396

What to know first

chapter 13, Named types · `layout`, `view` and field access markers
chapter 20, Allocators and fixed memory · fixed windows on machines without an operating system
chapter 29, Meeting C · places that cannot be checked are confined by mark, right and effects line

Looking back

What were the three things an op calling C had to have in chapter 29, and whom did each tell?

> **A.** The `unsafe` mark (the person), `cap c` (the processor) and the effects line (the caller). This chapter confines device registers and machine instructions the same way. Only the capability and effect names change.

The need for this chapter, and its context

The lowest place of a systems language is firmware reading and writing device registers directly, without an operating system. Defects here are quiet and fatal. A compiler deletes a register read that "looks useless", a write-only register is read and moves the device, or an interrupt handler is called like an ordinary function and runs on the wrong stack. Lowent stops these at translation. This chapter gathers how the tools set up in earlier chapters — capabilities, effects, layout, fixed windows — are used on hardware.

By the end of this chapter

You will write a device's register map as a struct with `mmio <address>`, and learn access through `read_volatile` and `write_volatile`. You will see that `cap mmio`, the `device` effect and the read-only and write-only markers are enforced at translation, and why a device cannot be received by value. You will also make interrupt handlers with the `vector` clause, decide the effects a machine supports with `build tier`, and confine machine instructions with the `asm` clause.

The questions this chapter answers

1 C adds the `volatile` qualifier. What is different?

30.1 The register map

examples/ch30/gpio.low

```
module gpio_demo .
rem run: drive 0 [0,0,0,0,7,0,0,0,0,0,0,0]

build tier t1 .

struct gpio do
  mmio 0x40020000 .
  moder u32 rw .
```

```
  idr u32 ro .
  bsrr u32 wo .
end

unsafe proc drive
  input dev cap mmio .
  input regs mut slice u8 .
  output u32 .
  effects device unsafe .
do
  var g gpio be view gpio regs .
  write_volatile g moder 2 .
  write_volatile g bsrr 1 .
  return read_volatile g idr .
end
```

Output

```
$ lowentc --run drive gpio.low 0 [0,0,0,0,7,0,0,0,0,0,0,0]
drive(0, [2,0,0,0,7,0,0,0,1,0,0,0]) = 7
  arg1 (written) = [2,0,0,0,7,0,0,0,1,0,0,0]
```

- `struct gpio do mmio 0x40020000 . … end` is the device's **map**. The number after `mmio` is the start address, and the fields become registers in order. There is no new word; a struct just gained a clause.
- `rw`, `ro` and `wo` after a field are its access.
- `read_volatile` and `write_volatile` reach a register. The processor **does not merge, delete or reorder** these accesses. Reading a device register is itself work and can change state.
- To reach registers, receive `cap mmio` and write `effects device`. Hardware access without a capability is exactly the ambient-authority problem seen with allocators.
- `build tier t1 .` declares that this module is for a small machine (covered below).

On the VM this **really runs**. Instead of a device, a byte buffer handed over by the caller takes the registers' place, and `view gpio regs` lays the map over it. In the argument shown in the result, `[2,0,0,0,7,0,0,0,1,0,0,0]`, you can see 2 written to `moder`, 1 to `bsrr`, and the 7 in `idr` read back. The first `0` in the run arguments is a placeholder filling the `cap mmio` position.

Laying the map over the buffer looks like this. Fields sit 4 bytes apart in declaration order.

```
address      field  access   in the VM (one u32 = 4 bytes)
0x40020000   moder  rw       [ 2 0 0 0 ]  ◄── write_volatile g moder 2
0x40020004   idr    ro       [ 7 0 0 0 ]  ──► read_volatile g idr = 7
0x40020008   bsrr   wo       [ 1 0 0 0 ]  ◄── write_volatile g bsrr 1
```

Q. C adds the `volatile` qualifier. What is different?

A. C's `volatile` is a property of a variable and easy to forget. Leave it off and the optimiser may delete reads without warning. In Lowent, reaching a register is a named **operation**, `read_volatile` or `write_volatile`, and the head of an op using it carries the `device` effect and `cap mmio`. Whether a read may be deleted is decided by the operation's name, not by remembering a qualifier on a variable.

30.2 Access is enforced at translation

Writing a read-only register is rejected.

examples/ch30/ro_write.low

```
module ro_write .
rem expect: E-MMIO-PERM

struct gpio do
  mmio 0x40020000 .
  moder u32 rw .
  idr u32 ro .
end

unsafe proc poke input dev cap mmio . input regs mut slice u8 . output u32 . effects dev
ice unsafe .
do
  var g gpio be view gpio regs .
  write_volatile g idr 1 .
  return 0 .
end
```

Output

```
$ lowentc --check ro_write.low
ro_write.low:13:0 E-MMIO-PERM: this register is READ-ONLY (`ro`) — writing it is a compi
le error, not a runtime surprise (RFC-0042 D3). The device says what it will accept; the
 type says it back
```

Reading a write-only register is the same. Reading one yields garbage or the read itself moves the device. That read is not useless but **wrong**. Once you have said what a device accepts, the type says it back.

Nor can a device be received by value.

examples/ch30/byvalue.low

```
module byvalue .
rem expect: E-MMIO-BYVALUE

struct gpio do
  mmio 0x40020000 .
  moder u32 rw .
end

unsafe proc setup input dev cap mmio . input g gpio . output u32 . effects device unsafe
 .
do
  write_volatile g moder 2 .
  return 0 .
end
```

Output

```
$ lowentc --check byvalue.low
9:0 E-MMIO-BYVALUE: an `mmio` register block cannot be a by-value parameter. A struct pa
```

```
rameter is COPIED, and copying a device means reading EVERY register at once — including
 the `wo` ones, whose reads D3 promises to reject at compile time, at a moment the progr
am never wrote. A register block is a DEVICE, not a value: pass the view instead (RFC-00
42 D3 · §8-4)
```

Struct parameters are copied. Copying a device means reading every register at once, write-only ones included, and writes to the copy never reach the device. A program written that way does nothing and says nothing. So translation stops it. Groups of registers are handled by laying them over a slice with `view`. Trying to open an absolute address on a machine with an operating system is rejected with `E-MMIO-NOHOST` — that address is not a device.

30.3 Interrupt handlers

Adding a `vector <number> .` clause to an op makes it an interrupt handler. Urgency is written with `priority <number> .`.

examples/ch30/isr.low

```
module isr .
rem flags: --target cortex_m

proc on_exti
  vector 6 .
  priority 2 .
  output void .
  effects device .
do
  return .
end
```

Output

```
$ lowentc --check isr.low
== check: ok ==
```

This example is checked with `--target cortex_m`. The machine calls interrupt handlers, so they keep four rules.

Rule	When broken	Why
Nobody calls it	`E-ISR-CALLED`	Called from our code it runs on the wrong stack and priority
No parameters	`E-ISR-PARAMS`	The machine passes no arguments
Returns nothing	`E-ISR-OUTPUT`	There is nobody to return to
Writes `effects device`	`E-ISR-EFFECT`	It exists because of a device

Table 30.1 — What an interrupt handler must keep

examples/ch30/isr_called.low

```
module isr_called .
rem expect: E-ISR-CALLED
```

```
proc on_exti
  vector 6 .
  output void .
  effects device .
do
  return .
end

proc impatient output void . effects device .
do
  on_exti .
end
```

Output

```
$ lowentc --check isr_called.low
isr_called.low:14:0 E-ISR-CALLED: nobody calls an interrupt handler — the HARDWARE does.
 Calling it from Lowent code runs it on the wrong stack, at the wrong priority, with int
errupts in the wrong state (RFC-0042 D5)
```

State shared by interrupt handlers and ordinary code moves by the discipline of priorities and queues. That discipline is a library's job, not the language's. The `spsc` ring buffer, through which one producer and one consumer pass values without locks, is used there (chapter 34).

30.4 What a machine supports — `build tier`

`build tier <name> .` says what this machine **supports**. Each tier fixes the effects that may be used.

Tier	Which machine	Newly supported effects
`t0`	Very small machine, no operating system	`none unsafe panic state wait cancel`
`t1`	Small machine	The above plus `io device`
`t2`	Real-time operating system	The above plus `alloc lock atomic blocking concurrent`
`t3`	Machine with an operating system	The above plus `heap page_fault detach` — that is, everything

Table 30.2 — Tiers and the effects they support

examples/ch30/tier.low

```
module tier .
rem expect: E-TIER-EFFECT

build tier t1 .

proc scratch input al cap allocator . output u64 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 16 .
  guard is_some g . else return 0 .
  return len (some_value g) .
end
```

Output

```
$ lowentc --check tier.low
tier.low:6:0 E-TIER-EFFECT: this effect is not available at the declared concurrency TIE
R. The tier says what the HARDWARE CAN CARRY — a thread pool on an ATtiny is not a runti
me problem, it is a COMPILE ERROR (RFC-0039 D4). And because effects PROPAGATE along the
 call chain, the refusal lands where you CALL it, not at link time
```

A `t1` machine was declared unable to support allocation (`alloc`), so it is rejected. This is not a matter of slowing down at run time; it simply cannot be loaded onto that machine. Effects rise along the call chain, so the rejection also happens at calling sites, not only where the effect first arises. Without a tier it is `t3`, and nothing is blocked.

Tier (what is supported) and profile (chapter 25 — which concurrency arrangement is used) are different axes. There are small machines with operating systems and large ones without.

> **A common misconception. Giving `--target cortex_m` sets the tier automatically**
>
> The target (`--target`) decides machine code and `machine.*` constants (whether there is a heap, and so on), which is why the heap is rejected by target alone (chapter 18). A tier is a **promise** written in the source. On the same target, writing `t0` can block even input/output. Only the one who writes a promise carries it, and the processor does not invent promises nobody wrote.

30.5 Machine instructions — `asm`

Some things portable code cannot reach: privileged instructions, system calls, exact cycle counts. There, an op's body is written in machine instructions.

examples/ch30/asm.low

```
module asm_add .

unsafe proc add_native
  input k cap machine .
  input a u64 .
  input b u64 .
  output u64 .
  effects unsafe .
  asm x86_64 .
    reg a .
    reg b .
    out reg r .
    clobber flags .
    options pure nomem nostack .
do
  text ASM
    mov {a}, {r}
    add {b}, {r}
  ASM
end
```

Output

```
$ lowentc --check asm.low
== check: ok ==
```

- The `asm x86_64 .` clause fixes the machine. Assembly for a machine other than the one being built for is rejected. The processor does not pretend it is portable.
- `reg a .` loads input `a` into a register and `out reg r .` receives the result. `clobber flags .` states what is clobbered, and `options …` states promises made to the processor.
- The body is a single `text ASM … ASM` heredoc. Assembly and ordinary code cannot be mixed.
- There are four confinements — the `unsafe` mark, `cap machine`, `effects unsafe` and the machine name.

The processor cannot read inside the template. But the operand list and the template's `{name}`s are two expressions of the same thing, so each is checked against the other.

examples/ch30/asm_unbound.low

```
module asm_unbound .
rem expect: E-ASM-UNBOUND

unsafe proc add_native
  input k cap machine .
  input a u64 .
  input b u64 .
  output u64 .
  effects unsafe .
  asm x86_64 .
    reg a .
    reg b .
    out reg r .
    clobber flags .
do
  text ASM
    mov {a}, {r}
    add {c}, {r}
  ASM
end
```

Output

```
$ lowentc --check asm_unbound.low
asm_unbound.low:4:0 E-ASM-UNBOUND: the template names an operand the `asm` clause never
declared — the assembler would either reject it or, worse, read WHATEVER register happen
s to be there
```

If the template names an undeclared `{c}`, the assembler rejects it or, worse, just reads whatever register was there. Conversely, a declared operand the template does not use is rejected too (`E-ASM-UNUSED`). `options pure` promises "no side effects", so the processor may trust it and delete or merge calls; if the effects line says it touches input/output or devices, both cannot be true, so it is rejected (`E-ASM-OPTLIE`). The VM cannot run machine instructions and says so (`E-VM-ASM`). It runs only natively.

30.6 The absorbing boundary — where `unsafe` stops

The four guards of the previous section carry a price: `effects unsafe` **travels with every call.** Write one line of assembly and the op that calls it must declare `unsafe`, and so must its caller, all the way up. That is why machine instructions were never used in the standard library (measured 2026-09-23: zero `asm` in `lib/`'s sixty-four modules).

The travelling is right — if the caller does not know about work the processor cannot see, the effect row is a lie. What was missing was **a place that takes responsibility.**

```
export proc add2 input a u64 . input b u64 . output u64 . effects none .
  absorbs machine k .          rem it stops here; `k` is this body's `cap machine`
  reference add2_soft .        rem a pure version that must give the same answer
  why "it adds two registers and touches no memory (options pure nomem nostack)." .
  requires ge a 0 .
do
  return asm_add2 k a b .
end
```

- Callers of this op **write nothing.** That is the whole value of absorption.
- `absorbs machine <name>` makes that name a `cap machine` inside the body. This is where that right is born — `machine` is not in the list an entry point may receive.
- Only `machine` may be absorbed. Capabilities that touch the world (io, C, heap) may not (`E-ABSORB-SCOPE`): minting one would create authority the caller cannot see.

As a call chain, the difference looks like this.

```
without absorbs                          with absorbs
main      effects unsafe  ▲              main      (nothing to write)
add2      effects unsafe  │ spreads up   add2      absorbs machine k ◄ stops here
asm_add2  effects unsafe  │              asm_add2  effects unsafe
```

Leave a prerequisite out and it is refused: a non-empty effect row is `E-ABSORB-IMPURE`, a missing reference implementation `E-ABSORB-NOREF`, a missing `requires` `E-ABSORB-NOCONTRACT`, an empty `why` `E-ABSORB-NOWHY`.

Who may absorb is the manifest's call. Absorption says a human vouches, so the source that wants the right cannot grant it to itself. Only a module named in `pkg.low` with `build absorb <module> .` may use the clause; otherwise it is `E-ABSORB-PLACE`. With no manifest at all, nobody may absorb.

The tool names the places. `lowentc --absorbs f.low` prints, one line each, which op absorbed what and why. A region the tool cannot see is made visible, not hidden.

```
  add2                    absorbs machine as `k`  ref=add2_soft  line 40
      why: it adds two registers and touches no memory…
absorbs: 1 op(s) stop `unsafe` here
```

Timing is written down, not checked. Whether the code takes value-dependent time is something the tool cannot verify. So the absorb registry has a timing column a human fills in, and leaving it empty fails the gate — «unknown» is a valid answer.

> **Why demand a reference implementation**
>
> Without a pure version that computes the same answer there is no way to tell «right» from «consistently wrong». It happened here: a GHASH rewritten with machine instructions passed

> the seal-then-open test while computing a different product entirely. Sealing and opening share the code, so the round trip holds as long as it is self-consistent. What caught it was comparing against the plain computation.

30.7 Using what the machine has — `--hw`

Whether to use the instructions is the builder's choice.

What you pick	What happens
`--hw none` (default)	everything in plain code; stands on any machine
`--hw pclmul,aes,sse2,avx2`	emitted **assuming** those instructions; will not run where they are missing
`--hw auto`	carry them all and choose **once at start**; stands anywhere, fast where the instructions exist. Includes VAES and VPCLMULQDQ (two blocks per ymm register) — on first use they are checked against the AES-NI path and left unused if they differ

Table 30.3 — What `--hw` chooses

What is carried is not only an instruction that replaces a computation. What `sse2` and `avx2` give is **width** — the room to put four or eight independent pieces of work side by side in one register, which is exactly the shape of ChaCha20's blocks. The rules are the same: the answer does not change, a machine that lacks the set cannot carry it, and the VM always runs the plain code.

- The answer is the same either way. What differs is speed and timing behaviour — a computation that reads tables reads at a value-dependent place; these instructions do not.
- Asking for an instruction set the target does not have is refused (`E-HW-TARGET`). There is no silent fallback: the builder must know what will run.
- The VM always runs the plain code, so this repository's oracle — «do the VM and native agree?» — is also the test for the machine path.

30.8 Common mistakes

> **Counter-example. Reading a write-only register to check what was just written**

examples/ch30/mistake_woread.low

```
module mistake_woread .
rem expect: E-MMIO-PERM

struct gpio do
  mmio 0x40020000 .
  moder u32 rw .
  idr u32 ro .
  bsrr u32 wo .
end

unsafe proc last_set input dev cap mmio . input regs mut slice u8 . output u32 . effects
```

```
 device unsafe .
do
  var g gpio be view gpio regs .
  rem ✘ reads a write-only register to check what was just written
  return read_volatile g bsrr .
end
```

Output

```
$ lowentc --check mistake_woread.low
mistake_woread.low:15:0 E-MMIO-PERM: this register is WRITE-ONLY (`wo`) — reading it is
a compile error. A wo register often reads as garbage (or has a read side effect), so th
e read is not merely useless: it is wrong
```

For an ordinary variable, writing and reading back is a good habit, but reading a write-only register yields garbage or makes the device do something. Hence `E-MMIO-PERM`. If you need the value you just wrote, keep it in a local before writing, and check the device's real state through the read register the datasheet defines (here, `idr`).

Counter-example. Giving an interrupt handler parameters

examples/ch30/mistake_isrparams.low

```
module mistake_isrparams .
rem flags: --target cortex_m
rem expect: E-ISR-PARAMS

rem ✘ tries to receive which pin fired as an argument --- the hardware passes no argumen
ts
proc on_exti
  vector 6 .
  input pin u32 .
  output void .
  effects device .
do
  return .
end
```

Output

```
$ lowentc --check --target cortex_m mistake_isrparams.low
mistake_isrparams.low:6:0 E-ISR-PARAMS: an interrupt handler takes NO parameters — the H
ARDWARE calls it, and hardware does not pass arguments. Shared state goes through the pr
iority/queue discipline (RFC-0039 D1c), not through a parameter list
```

The hardware passes no arguments when it calls a handler. Which pin fired is learned inside the handler by reading the device's status register. Values shared with ordinary code travel through the priority and queue discipline (the `spsc` ring buffer). Hence `E-ISR-PARAMS`.

Counter-example. Leaving an interrupt handler's effects line empty

examples/ch30/mistake_isreffect.low

```
module mistake_isreffect .
rem flags: --target cortex_m
rem expect: E-ISR-EFFECT

rem ✘ the effects line is left out because there is nothing to do yet
proc on_exti
  vector 6 .
  output void .
do
  return .
end
```

Output

```
$ lowentc --check --target cortex_m mistake_isreffect.low
mistake_isreffect.low:6:0 E-ISR-EFFECT: an interrupt handler must declare `effects devic
e` — it exists because the device asked for it
```

Even with nothing to do yet, an interrupt handler exists because of a device. Writing `effects device .` keeps in the head the fact that this op is device-side code, and the tier (`build tier`) and capability checks follow that fact. Without it, `E-ISR-EFFECT`.

Counter-example. Spelling the `asm` machine name the way another toolchain does

examples/ch30/mistake_asmtarget.low

```
module mistake_asmtarget .
rem expect: E-ASM-TARGET-UNKNOWN

unsafe proc add_native
  input k cap machine .
  input a u64 .
  input b u64 .
  output u64 .
  effects unsafe .
  rem ✘ the machine is spelt as another toolchain spells it (`aarch64`) --- this process
or's name is `arm64`
  asm aarch64 .
    reg a .
    reg b .
    out reg r .
do
  text ASM
    add {r}, {a}, {b}
  ASM
end
```

Output

```
$ lowentc --check mistake_asmtarget.low
mistake_asmtarget.low:4:0 E-ASM-TARGET-UNKNOWN: this `asm` clause names something that i
s not a build target. The names are a CLOSED set — the target table is the authority (x8
6_64 · arm64 · cortex_m · riscv64 · mips_be). A name outside it matches NO build, so the
```

```
 op would be silently dropped or silently mis-built; both are what this RFC exists to pr
event
```

GCC and LLVM say `aarch64`, but this processor's target name is `arm64`. The names are a closed list (`x86_64`·`arm64`·`cortex_m`· `riscv64`·`mips_be`), and a name outside it matches no build, so the op would never be built. Instead of skipping it silently, the tool stops with `E-ASM-TARGET-UNKNOWN`.

Counter-example. Reading and writing registers as ordinary fields

examples/ch30/mistake_plainfield.low

```
module mistake_plainfield .
rem expect: E-MMIO-PLAIN

struct gpio do
  mmio 0x40020000 .
  moder u32 rw .
  idr u32 ro .
  bsrr u32 wo .
end

unsafe proc drive input dev cap mmio . input regs mut slice u8 . output u32 . effects de
vice unsafe .
do
  var g gpio be view gpio regs .
  rem ✘ writes and reads registers as ordinary fields --- these read as accesses that ma
y be merged or removed
  set (field g moder) 2 .
  return field g idr .
end
```

Output

```
$ lowentc --check mistake_plainfield.low
mistake_plainfield.low:15:0 E-MMIO-PLAIN: this is a DEVICE register, and this reads/writ
es it like ordinary memory. An ordinary access is one the translator may merge or delete
 (two reads of the same field become one; a store nothing reads goes away) — for a devic
e the access ITSELF is the work, so deleting it makes the hardware do the wrong thing. W
rite `read_volatile <block> <reg>` or `write_volatile <block> <reg> <value>`, which say
exactly once, in the written order (RFC-0042 D1). The `ro`/`wo` permission is checked th
ere too
```

`set (field g moder) 2` and `field g idr` are refused (`E-MMIO-PLAIN`). Ordinary field access is an operation the processor may merge or remove — two reads of one register becoming one read, or a store nothing reads going away, change nothing in ordinary memory. On a device the access itself is the work, so the answers change. For device registers write `read_volatile` and `write_volatile`, which is also where the access modes (`ro`·`wo`) are checked.

30.9 This chapter's syntax at a glance

Shape	Meaning	Why

`struct gpio do mmio 0x40020000 . moder u32 rw . … end`	a device's register map	one clause on a struct, no new words
`rw · ro · wo`	access modes — enforced at translation	violations: `E-MMIO-PERM`
`read_volatile g idr · write_volatile g moder 2`	access that is never merged or removed	reading is itself an action
`input dev cap mmio . + effects device`	device capability and effect	no hardware access without authority
`var g gpio be view gpio regs .`	lay the map over bytes	taking a device by value: `E-MMIO-BYVALUE`
`proc on_exti vector 6 . priority 2 . output void . effects device .`	an interrupt handler	calling it: `E-ISR-CALLED` · arguments: `E-ISR-PARAMS`
`build tier t1 .`	the tier of effects this machine can bear	what cannot be carried is stopped at translation
`asm x86_64 . reg a . out reg r . clobber flags . options pure .`	head of a body written in machine code	confined by `unsafe`, `cap machine`, effects line, machine name
`text ASM … {a} … ASM`	the template — checked against the operands	`E-ASM-UNBOUND` · `E-ASM-UNUSED` · `E-ASM-OPTLIE`

Table 30.4 — Hardware syntax — shape · meaning · why it looks this way

Recap

A struct with `mmio <address>` is a device's map, and `read_volatile` and `write_volatile` are accesses never merged or deleted. They need `cap mmio` and the `device` effect, the `ro` and `wo` markers are enforced at translation, and devices cannot be received by value. An op with a `vector` clause is an interrupt handler that can neither be called nor take arguments. `build tier` fixes the effects a machine supports. The `asm` clause confines machine instructions with `unsafe`, `cap machine`, an effects line and a machine name, and checks operands and template against each other.

31 Building and testing — packages, configuration, tests, cross-checks

In this chapter

31.1 The manifest and the project 398
31.2 Dependencies are pinned by hash 399
31.3 A different program per build — `build option` and `config` 400
31.4 Tests 402
31.5 How the compiler is verified 404
31.6 Asking where it is slow 405
31.7 Common mistakes 405

31.8 This chapter's syntax at a glance 408

What to know first

chapter 2, A first program · the modes of `lowentc` and its two back ends
chapter 21, Modules · there is no search path
chapter 26, Tasks and channels · `schedule explore_interleavings` tests

Looking back

In chapter 2, whose defect was it when the VM and native code give different output? And what does a quiet `--check` **not** guarantee?

> **A.** The compiler's, because the two back ends start from the same intermediate representation. A quiet `--check` goes only as far as the head's promises and the body not disagreeing; it does not guarantee that tests pass or that the program is right. This chapter covers the tools filling the gap — tests, configuration and cross-checks — and how to build a project.

The need for this chapter, and its context

This is the last chapter of Part VIII. Earlier chapters ran `--check` and `--run` on one file. Real projects have a manifest, configurations that differ per build, tests and dependencies. Lowent's tools apply the same principle to all of them — what is written is checked, defaults are visible in the source, and nothing changes quietly. And the compiler itself is verified by the same principle.

By the end of this chapter

You will learn to build a project with the `pkg.low` manifest and `lowentc run` and `build`. You will see that `build option` and `config` build different programs per build while switched-off branches are still checked. You will also see what `test` blocks look like when they pass and fail, how cross-checking the two back ends and contracts verifies the compiler, and how dependencies are pinned by content hash.

The questions this chapter answers

1 If there are tests, must contracts be written too?

31.1 The manifest and the project

Put `pkg.low` at the project root and the entry point in `src/main.low`.

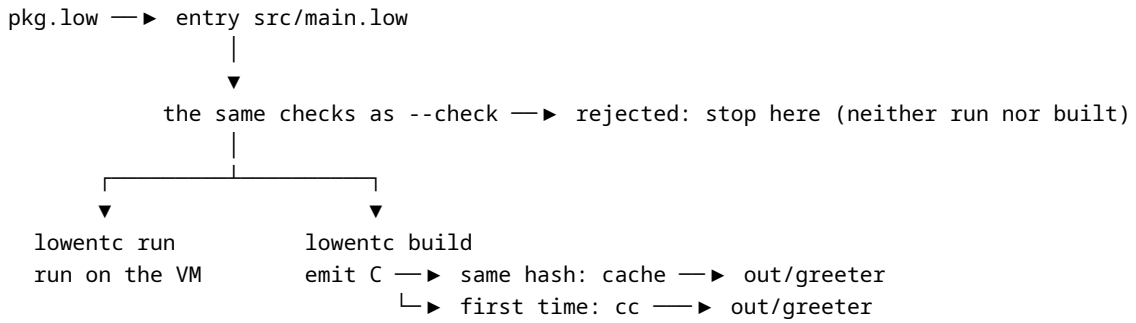
```
package name "greeter" .
package version "0.1.0" .
package license "MIT" .
```

The manifest is a few flat declarations. The keys are a closed set and `version` is checked against semantic versioning. This file exists so that tools and people can know a project's identity from

it alone without reading all the source, so here too what is written is checked. If it cannot be trusted when read, it may as well not exist.

```
$ lowentc run
hello from a package
main() = 0
$ lowentc build
built …/out/greeter
$ ./out/greeter main
hello from a package
```

`lowentc run` finds the manifest and runs the entry point on the VM, and `lowentc build` emits C and builds an executable in `out/`. Builds are cached by the content hash of the emitted C, so unchanged code is not compiled again. The manifest is found by walking upwards from the first file's folder.



Both commands pass the same checks as `--check` **before** running or building. A program the language rejects neither runs nor becomes an executable. `run` has done so since 2026-09-16, `build` since 2026-09-26 — before that, `build` skipped the checks and built even a program with a mistake like `return true .` (found by measuring while this chapter was revised).

31.2 Dependencies are pinned by hash

`lowentc add <name> <place>` writes a dependency into the manifest **with a hash pin**. `--lock-write` records in a lock file what is being built against now, and `--lock <file>` refuses to build if the bytes differ from that hash. Even with the same version number, changed bytes are a different dependency.

Checking authenticity is a separate layer. `lowentc key new`, `sign` and `verify` make and check ed25519 detached signatures. `verify` splits its answer into four layers — integrity (do hash and pin match), authenticity (signed with a trusted key), transport and access — because green in one layer does not mean green in another. The only environment variable read to find settings is `HOME`, and the precedence of settings (command line > project > user > global) is written in the help. This is to reduce places where behaviour is changed secretly outside the source. A few other switches whose names start with `LOW` exist, all off by default and meant for testing and migration — the fault injector `LOW_HOST_FAULT` (chapter 28), the rollback door `LOWENT_ALLOW_GLUED_FIELD` that briefly re-admits the removed glued-dot field access, and the step limit `LOWENT_ORACLE_BUDGET` of the oracle tools. There is no everyday reason to set them, and setting one shows on the command line.

This is how `verify` reports its four layers separately (measured on an unsigned file).

```
[3 integrity]     BLAKE3 a19e7c93…      are these bytes the pinned ones
[2 authenticity]  unsigned — LOCAL       who published it --- a trusted key's signature
[1 transport]     TLS (curl's job)       who the server is --- not what the bytes are
[4 access]        tokens in tool config  may you get in --- separate from integrity
```

Each layer is looked at on its own even when another is green. A successful TLS connection means the server is the right one, not that the bytes received are.

31.3 A different program per build — `build option` and `config`

examples/ch31/knobs.low

```
module knobs .
rem run: tick_rate
rem run: cpus

build option smp bool default true .
build option hz choice 100 250 1000 default 250 .
build option maxcpu int default 64 .

fn tick_rate output u64 .
do
  if config smp . do
    return config hz .
  end
  return 100 .
end

fn cpus output u64 .
do
  return config maxcpu .
end
```

Output

```
$ lowentc --run tick_rate knobs.low
tick_rate() = 250
$ lowentc --run cpus knobs.low
cpus() = 64
```

`build option <name> <kind> …` declares a knob. There are three kinds: `bool`, `int` and `choice`. `config <name>` reads its value as a **translation-time constant**. Without a configuration file, the declaration's default is used.

The same source is built with a different configuration. A configuration file is a set of `name value` lines.

```
smp false
maxcpu 8
```

examples/ch31/knobs_small.low

```
module knobs_small .
rem flags: --config small.config
rem run: tick_rate
rem run: cpus
```

```
build option smp bool default true .
build option hz choice 100 250 1000 default 250 .
build option maxcpu int default 64 .

fn tick_rate output u64 .
do
  if config smp . do
    return config hz .
  end
  return 100 .
end

fn cpus output u64 .
do
  return config maxcpu .
end
```

Output

```
$ lowentc --config small.config --run tick_rate knobs_small.low
tick_rate() = 100
$ lowentc --config small.config --run cpus knobs_small.low
cpus() = 8
```

With `smp` switched off, `tick_rate` gives 100 and `cpus` gives 8. Once the value is fixed, switched-off branches do not remain in the output. The cost is zero.

```
build option smp bool default true .       ← the source declares the knob
small.config:  smp false                   ← the config file gives a value (default if absent)

if config smp . do return config hz . end  ← the branch when smp is on
return 100 .                               ← the branch when it is off

translation  both branches are parsed and type-checked
output       only the chosen branch stays (the off branch costs 0)
```

There is a decisive difference from C's `#ifdef`. **Switched-off branches are still parsed and type-checked.** In C, code for combinations nobody switches on rots unread, and in kernel-scale projects "that option combination does not even build" happens. Here what folds away is **code emission**, not checking.

A knob declared but read by no code is rejected.

examples/ch31/unused.low

```
module unused_opt .
rem expect: E-OPT-UNUSED

build option trace bool default false .

fn answer output u64 .
do
  return 42 .
end
```

Output

```
$ lowentc --check unused.low
unused.low:4:0 E-OPT-UNUSED: this build option is declared and NO code reads it (`config
 <name>`). It shows up in the configuration, the user turns it off — and nothing happens
. A knob that does nothing is worse than no knob: it is a decision that LOOKS like it wa
s made
```

Such a knob appears in the configuration, a user switches it off, and nothing happens. It looks like a decision but is not one. A configuration file giving a value not among the choices (`E-CONFIG-TYPE`) or naming a knob that does not exist (`E-CONFIG-UNDEF`) is rejected too.

> **In practice. A knob that could not be switched off**
>
> The specification's example makes the `hz` knob depend on `smp` (`depends smp`). The rule is that if the dependent is on while what it depends on is off, no such build exists, so it is rejected. Running that example while writing this book, a `choice` knob always had a value and read as "on", so every configuration switching `smp` off was rejected with `E-CONFIG-DEPENDS`. The development repository's test looked only at running with that configuration, not at checking, and missed the discrepancy. That is why this chapter's example leaves out `depends`. The lesson that a green light says nothing when what the test sees differs from what users do applies here too.

31.4 Tests

`expect` inside a `test <name> do … end` block is an assertion. `--test` runs every test.

examples/ch31/tests_clause.low

```
module tests_clause .
rem test

fn clamp8 input v u64 . output u8 .
do
  return narrow_sat u8 v .
end

test clamp_keeps_small
do
  expect eq (clamp8 7) 7 .
end

test clamp_saturates
do
  expect eq (clamp8 1000) 255 .
end
```

Output

```
$ lowentc --test tests_clause.low
  [PASS] clamp_keeps_small
  [PASS] clamp_saturates
== tests: 2 run, 2 passed, 0 FAILED ==
```

When a test fails, it looks like this.

examples/ch31/failing.low

```
module failing .
rem test-fail

fn clamp8 input v u64 . output u8 .
do
  return narrow_wrap u8 v .
end

test clamp_saturates
do
  expect eq (clamp8 1000) 255 .
end
```

Output

```
$ lowentc --test failing.low
  [FAIL] clamp_saturates
         E-TEST-FAIL: an `expect` in this test is FALSE — the test failed (this is not a
 contract violation: it is the test telling you the code is wrong)
== tests: 1 run, 0 passed, 1 FAILED ==
```

`E-TEST-FAIL` is a different diagnostic from a contract violation. A contract violation is **code breaking its own promise**; a test failure is **a test saying the code is wrong**. What to fix differs. Here `narrow_wrap` must become `narrow_sat`. And the processor does not optimise `expect` away. A vanished test is a test that did not run.

Concurrent code uses `test <name> schedule explore_interleavings . do … end` to run every possible ordering of flows and see whether the answers agree (chapter 26). When there are many cases, `limit <number>` sets a ceiling.

An op head also has clauses for documentation and tests.

examples/ch31/clauses.low

```
module clauses .
rem run: first_two [65,66]
rem test

enum parse_error do
  too_short .
end

fn first_two
  rem lowdoc is a description for people; tools extract it as documentation
  lowdoc "Add the first two bytes; refuses input shorter than two." .
  input data slice u8 .
  output result u64 parse_error .
  errors too_short lt (len data) 2 .
  rem tests names the ops that test this op; the tool checks that those names exist
  tests first_two_ok first_two_short .
do
  guard ge (len data) 2 . else return error too_short .
  return ok (add (widen u64 (index data 0)) (widen u64 (index data 1))) .
end
```

```
fn first_two_ok output bool . do
  let r result u64 parse_error be first_two "AB" .
  return is_ok r .
end

fn first_two_short output bool . do
  let r result u64 parse_error be first_two "A" .
  return is_error r .
end

rem the test block calls the testing ops; lowentc --test runs it
test first_two_cases do
  expect first_two_ok .
  expect first_two_short .
end
```

Output

```
$ lowentc --run first_two clauses.low [65,66]
first_two([65,66]) = ok 131
  arg0 (written) = [65,66]
$ lowentc --test clauses.low
  [PASS] first_two_cases
== tests: 1 run, 1 passed, 0 FAILED ==
```

- `lowdoc "…" .` is a description for people. Unlike a comment (`rem`), it stays attached to the op as documentation that tools extract. For a long one, write several lines with `lowdoc text DOC … DOC .` (chapter 3).
- `tests first_two_ok first_two_short .` lists **the names of the ops** that test this op, not the names of `test` blocks. The tool checks that those names exist and refuses with `E-CONTRACT-UNDEF` if they do not. If a test is renamed or removed, the head tells you.
- `errors too_short lt (len data) 2 .` is a failure with a condition. The two testing ops check the success arm and the failure arm, and the `test` block ties them together with `expect`.

> **Q. If there are tests, must contracts be written too?**
>
> **A.** They catch different things. A test checks answers on inputs the author picked; a contract checks that promises hold on every call. And contracts become **material for making tests**. The development repository's contract cross-check tool uses `requires`, `ensures` and `errors` as the verdict without writing expected output separately, and generates boundary-value inputs. Write contracts honestly and tests grow for free.

31.5 How the compiler is verified

The proofs (Part X) are about a **model**. Whether the actual compiler follows that model must be confirmed separately. There is one principle — **make the same thing in two ways, and treat a different answer as a defect.**

Method	What is compared with what	What it catches

Back-end cross-check	VM run ↔ native run	Places where the two back ends answer differently
Contract cross-check	Written contracts ↔ actual run results	Places where contracts differ from the facts
Analysis self-check	Indices believed “safe” ↔ actual indices	Places where the analysis is wrong (`E-VM-ANALYSIS`)
Certificate recheck	Grounds for removing a check ↔ an independent checker’s arithmetic	Places where rules were misapplied
One-line changes	A program ↔ the program with one fact removed	Broken relations, even without knowing the right answer

Table 31.1 — Ways of checking the compiler against something

Such checks show that defects **exist** but cannot show that they **do not**. Two implementations giving the same answer are not thereby both right. Absence is the job of proofs. What you can run yourself in the public repository is `make check` in `impl/`. It covers unit tests, checking the whole standard library, VM/native cross-checks over many ops, and the diagnostic codes of programs that must be rejected, all at once. This book’s verification script applies the same cross-check to every example — and while this book was written, that cross-check exposed several discrepancies in the compiler.

> **A common misconception. A green light means it was checked**
>
> A green light speaks only about **what it saw**. An operation not on the cross-check list shows neither green nor red. So the development repository makes the cross-check **count** and report opcodes it has never seen. “The knob that could not be switched off” above is the same: the test looked at running, not checking, so it was green. What is not counted is not managed.

31.6 Asking where it is slow

Native emission lowers ops by two paths. Ops whose types are settled lower naturally to C integers and arrays; the rest stay on a slow path running on tagged values. `--why-slow` names the ops left on the slow path and why.

```
$ lowentc --why-slow bounds.low
why-slow: 0 / 1 op(s) still on the tagged path
```

A model whose performance is not visible in signatures becomes knowledge outside the source. So the tool says it. `--no-fast` is a contrast switch lowering every op by the slow path, used to see whether both paths give the same answer.

31.7 Common mistakes

> **Counter-example. Misspelling a knob name in the config file**
>
> **examples/ch31/mistake_configtypo.low**

```
module mistake_configtypo .
rem flags: --config typo.config
rem expect: E-CONFIG-UNDEF

rem ✘ the config file spells `maxcpu` as `maxcpus`

build option smp bool default true .
build option hz choice 100 250 1000 default 250 .
build option maxcpu int default 64 .

fn tick_rate output u64 .
do
  if config smp . do
    return config hz .
  end
  return 100 .
end

fn cpus output u64 .
do
  return config maxcpu .
end
```

Output

```
$ lowentc --check --config typo.config mistake_configtypo.low
0:0 E-CONFIG-UNDEF: the config selects an option that is not declared. A setting nobody
reads changes nothing — and it LOOKS like it does
```

`maxcpus 8` is a setting nobody reads. Passing over it quietly would leave `maxcpu` at its default of 64 while the user believes the build used 8. So it stops with `E-CONFIG-UNDEF`. The line number `0:0` is because the mistake is in the config file, not in the source.

Counter-example. Giving a `choice` knob a value that is not one of its choices

examples/ch31/mistake_configvalue.low

```
module mistake_configvalue .
rem flags: --config badvalue.config
rem expect: E-CONFIG-TYPE

rem ✘ the config file gives `hz` the value 300, which is not one of its choices

build option smp bool default true .
build option hz choice 100 250 1000 default 250 .
build option maxcpu int default 64 .

fn tick_rate output u64 .
do
  if config smp . do
    return config hz .
  end
  return 100 .
end
```

```
fn cpus output u64 .
do
  return config maxcpu .
end
```

Output

```
$ lowentc --check --config badvalue.config mistake_configvalue.low
8:0 E-CONFIG-TYPE: the config picks a value this `choice` option does not offer
```

`hz` is one of 100, 250 and 1000. Given 300, the tool neither rounds to the nearest nor falls back to the default; it stops with `E-CONFIG-TYPE`. The choices also mean “tested with these values”. If you need a new value, add it to the declaration in the source.

A common misconception. An op that takes a capability cannot be called with `--run`

examples/ch31/runcap.low

```
module runcap .
rem run: say 5
rem run: say 0 5

rem `--run` FILLS capability positions when you leave them out (2026-09-20). The old pla
ceholder `0` still works too
proc say input out cap io . input n u64 . output u64 . effects io .
  requires le n 100 .
do
  return write_out out 1 "hi\n" .
end
```

Output

```
$ lowentc --run say runcap.low 5
hi
say(5) = 3
$ lowentc --run say runcap.low 0 5
hi
say(0, 5) = 3
```

That was true until 2026-09-20: capability positions counted among the arguments, so `say 5` was `E-VM-ARITY` and only `say 0 5`, with a placeholder `0`, ran. The cost was quiet: **witness programs** taking `cap io` or `cap allocator` ran on the native back end only, so the two-back-end oracle this book relies on was broken exactly there. The tool now FILLS capability positions — but only when arguments are missing — so both `say 5` and `say 0 5` run. The placeholder is still only a convenience of the tool: inside a program, passing a number where a capability belongs is refused with `E-CAP-FORGE` (chapter 16).

A common misconception. Branches switched off by the configuration are not checked

examples/ch31/dead_branch.low

```
module dead_branch .
rem flags: --config smp_off.config
rem expect: E-TYPE-RETURN

build option smp bool default true .

fn rate output u64 .
do
  if config smp . do
    rem this branch is switched off in this configuration --- it is checked all the same
    return true .
  end
  return 0 .
end
```

Output

```
$ lowentc --check --config smp_off.config dead_branch.low
dead_branch.low:11:0 E-TYPE-RETURN: the returned value does not match the op's `output`
— expected `u64`, found `bool`
```

With `smp` switched off, `return true .` never runs. Even so, this line, which returns a `bool` where a `u64` belongs, is rejected with `E-TYPE-RETURN`. Code inside C's `#ifdef` goes unread in combinations that are not enabled, but a switched-off `config` branch is parsed and type-checked; it is only left out of the output once the value is known. So whichever configuration you build, the other combinations do not rot.

31.8 This chapter's syntax at a glance

Shape	Meaning	Why
`package name "greeter" .` · `package version "0.1.0" .`	the `pkg.low` manifest	here, what is written is checked — a closed set of keys
`lowentc run` · `lowentc build`	run the project on the VM · build into `out/`	the manifest is found by walking upward
`lowentc add <name> <place>` · `--lock-write` · `--lock`	pin dependencies with hashes	same version, different bytes: a different dependency
`build option smp bool default true .`	a build knob — `bool`·`int`·`choice`	a knob nobody reads is `E-OPT-UNUSED`
`config smp` · `--config small.config`	read a knob as a translation-time constant · a config file	switched-off branches are checked too
`test <name> do expect <condition> . end` · `--test`	test blocks and assertions	failure is `E-TEST-FAIL` — not a contract violation
`test … schedule explore_interleavings limit <n> . do … end`	a test that runs every order	bugs of rare orders

`lowentc --run <op> <file> <args…>`	run one op on the VM	the tool fills capability positions — a rejected unit does not run
`--why-slow` · `--no-fast`	ops left on the slow path and why · everything on the slow path	the tool speaks about performance
`lowdoc "…" .` · `tests op1 op2 .`	documentation attached to the op · names of ops testing it	docs move with the op, and a missing test is reported by the head

Table 31.2 — Build and test syntax — shape · meaning · why it looks this way

Recap

`pkg.low` is a checked manifest, and `lowentc run` and `build` run and build the project. Dependencies are pinned by content hash, and authenticity is checked separately by signatures. `build option` and `config` build different programs per build while switched-off branches are still checked, and unread knobs are rejected. A failing `test` block is a different diagnostic from a contract violation. The compiler is verified against two back ends, contracts, analysis, certificates and one-line changes, and `--why-slow` says where it is slow.

Part IX — The standard library

This part tours the standard library. Detailed descriptions of each module are in Appendix E, one page each, and this part does not repeat them. Instead it sets out, with examples, **which module sits on which layer, which conventions they all follow, and where to look for a given job**.

The map (chapter 32) — the three layers of language, leaf and library, the criteria for entering the library, the layers of pure computation, storage and host, and the four conventions every module follows.

A tour by use (chapters 33–37) — text and encodings, containers and sorting, storage and handles, input/output, networking, time, randomness and cryptography, and the terminal. Each chapter actually runs a few representative modules and points to the rest in one line each.

While reading this part you will see one thing again and again. The library has no privileges, so the rules learned in the previous eight parts — contracts, effects, capabilities, ownership — apply as they are. Not a single new word was added. And the top of every module document always says **what it does not do**.

32 A map of the standard library

In this chapter

32.1 Three layers — language, leaf, library 412
32.2 What gets in 412
32.3 The map by layer 413
32.4 File names and module names 414
32.5 Conventions every module follows 415
32.6 The list of leaf ops — what lies beneath the modules 415
32.7 Common mistakes 418
32.8 This chapter's syntax at a glance 419

What to know first

chapter 21, Modules · standard library names are brought in with `use <name>` .
chapter 16, Capabilities · capabilities are handed over, not picked up
chapter 17, Designing failure · at the boundary, failure is returned as a value

Looking back

In chapter 21, does `use` look for a file name or a module name? Why did it say the difference matters in the standard library?

> **A.** The `module` declaration inside the file. In the standard library a few file names differ from module names (`lib/alloc.low` is `allocs`, `lib/str.low` is `strings`), so trying to import by file name leads astray. This chapter's first table is that list.

The need for this chapter, and its context

Part IX tours the standard library. Detailed descriptions of each module live in Appendix E, one page each, and this part does not repeat them. Instead it sets out **which module sits on which layer, which conventions they all follow, and where to look for a given job**. This first chapter is the map — the boundary between language and library, the criteria for entering the library, and the layers and shared conventions of modules.

By the end of this chapter

You will learn the question that separates the three layers of language, leaf and library. You will pick up the criteria (the charter) and maturity ladder for entering the standard library, and a map dividing modules into a layer that does not allocate, a layer needing storage and a layer needing capabilities. You will also see the shared conventions — the caller's buffer, failure returned as a value, all or nothing, ownership that must be settled — and the places where file names and module names differ.

The questions this chapter answers

1 Why leave as a library a feature that would be convenient in the language?

32.1 Three layers — language, leaf, library

Layer	Who makes it	Qualification
Language	The specification	Cannot be expressed as a library, or expressing it would hide a cost
Leaf	The processor	Reaches the operating system or hardware. Cannot be written in Lowent
Library	Anyone	Everything else. Written in Lowent

Table 32.1 — Three layers

There is one question deciding whether a piece is a leaf — “can this be written in Lowent?” If it can, it is library. Opening a file and receiving an integer handle reaches the kernel, so it is a leaf. Wrapping that handle in an owned value that cannot be forgotten can be written in Lowent, so it is library. A convenience op that reads a whole file is also library. **Convenience does not qualify something as a leaf.**

The standard library has no privileges. It gets the same rules as code the author writes — contracts, effects, capabilities. Here is the source of `strings.starts_with`.

```
export fn starts_with input s str . input prefix str . output bool . do
  guard le (len prefix) (len s) . else return false .
  return eq_str (subslice s 0 (len prefix)) prefix .
end
```

It reads with only what this book has taught. If there were a hidden passage only the library could use, a program’s guarantees would break the moment it passed through.

Q. Why leave as a library a feature that would be convenient in the language?

A. Adding a word or builtin to the language means every program must learn it and the processor must treat it specially. As a library it is checked by the same rules, and programs that do not use it do not have it. So the language accepts only **what cannot be expressed**. The standard library’s allocators, containers and file handles all passed this criterion and live in the library. Convenience is not a reason to enter the language.

32.2 What gets in

Things enter the library not **because they exist** but **on evidence**. The specification’s charter sets six measures.

Measure	Meaning
If it can be written in this language, write it in this language	Leaves only when using outside authority or when this language cannot express it

The signature is the ledger of cost and authority	Effects, capabilities, storage, allocation, ownership and completion are visible at the boundary
Separate computation from authority	Formatting vs. emitting, parsing vs. opening files, the next step of a random generator vs. getting entropy are different
Place along several axes	Authority, storage, execution tier and profile are not squeezed into one ladder
Keep what is inside small	Up to a portable base and a thin host adapter
Every module has evidence	Unit tests, rejection tests, cross-checks and failure tests

Table 32.2 — The library charter

Every module writes its **maturity** — `experimental` (merely written), `incubating` (has a decision document), `standard` (documented and named by regression tests) and `deprecated` (says what to move to). The value of the ladder is in the lower rungs, not the upper. Writing `experimental` writes "do not trust this yet" **in the source**. A rung is not a ranking of quality but the size of a promise.

The specification does not list the library's **contents**. What exists is authoritatively recounted from the source each time, and a document copying it diverges from the moment it copies. Three APIs that did not actually exist were once read as fact because they were in a table. The tables in this part were also made by looking at this edition's source, and detailed per-module pages are in Appendix E.

32.3 The map by layer

Layer	Character	Representative modules
L0 pure computation	`effects none` · the caller's buffer	`strings strbuf fmt utf8 utf16 unicode codec regex hash math random sortlib sortgen searchlib hashmap strmap spsc term` · cryptographic modules
L1 storage	Needs the allocation capability or borrowed bytes	`allocs pool shard budget segarena pagecache growvec vecgen mapgen nodelist segview wire flags lifemode`
L2 host	Needs capabilities (`cap io`, `file_system`, `tty`, `net`, `clock`)	`io outbuf files tty net clock`

Table 32.3 — Layers of the standard library

The remaining chapters of this part group by use: text and encodings (chapter 33), containers and sorting (chapter 34), storage and handles (chapter 35), input/output, networking and cryptography (chapter 36), and the terminal (chapter 37).

32.4 File names and module names

What `use` looks for is the `module` declaration inside the file. In a few places it differs from the file name.

Source file	Module name
lib/alloc.low	allocs
lib/str.low	strings
lib/vec.low	vecs
lib/sort.low	sortlib
lib/search.low	searchlib
lib/out.low	outbuf
lib/file.low	files

Table 32.4 — Module names that differ from file names

examples/ch32/names.low

```
module names .
rem run: is_get [71,69,84,32,47]
rem run: is_get [80,79,83,84]

use strings .

fn is_get input line slice u8 . output bool .
do
  return strings.starts_with line "GET " .
end
```

Output

```
$ lowentc --run is_get names.low [71,69,84,32,47]
is_get([71,69,84,32,47]) = 1
  arg0 (written) = [71,69,84,32,47]
$ lowentc --run is_get names.low [80,79,83,84]
is_get([80,79,83,84]) = 0
  arg0 (written) = [80,79,83,84]
```

`use strings .` resolves from the standard module location without `from`. Imported names are always called qualified by module, as in `strings.starts_with`. There is no glob import. Importing by file name finds no such module.

examples/ch32/wrong_name.low

```
module wrong_name .
rem expect: W-USE-EXTERNAL

use str .

fn is_get input line slice u8 . output bool .
do
  return true .
end
```

Output

```
$ lowentc --check wrong_name.low
wrong_name.low:4:0 W-USE-EXTERNAL: this module is not in the compilation unit — there is
 no module search path, so nothing here can confirm it exists. Pass the file that declar
es it and it WILL be checked (several .low files link into one unit)
```

The diagnostic is a warning rather than an error because there is no search path, so looking at this translation unit alone cannot tell whether the name exists somewhere (chapter 21). The moment the name is actually used, it is rejected.

32.5 Conventions every module follows

Convention	Meaning
Buffers belong to the caller	L0 modules do not allocate. They take output buffers and workspace as parameters. So reentrancy is free and the VM/native cross-check runs as is
Failure is a value	`option` (could not) or `result` (what failed). No traps, no silent truncation, no replacement characters
All or nothing	If there is not enough room, not a single byte is written. A half-written buffer is silently wrong output
Ownership must be settled	Things needing completion (`outbuf`'s remaining bytes, `files`' handles) are owned values, so forgetting them is rejected at translation

Table 32.5 — Shared conventions of the standard library

> **A common misconception. It is the standard library, so every module can be trusted equally**
>
> Maturity differs per module, and the top of each module document states what it **does not do**. For example, `tlssrv` performs the whole handshake but not yet transport, and `der` is a minimal parser extracting only public keys, not a certificate infrastructure (PKI). Reading the boundaries a module writes for itself before importing it is how this library is meant to be used.

32.6 The list of leaf ops — what lies beneath the modules

Library modules ultimately call **leaf** builtin ops provided by the processor. Normally you use the module and do not call the leaves directly — the module dresses them in conventions such as ownership, three-place answers and all-or-nothing. Still, knowing which module stands on what keeps you oriented when reading module documentation or building a new module.

Leaf op	What it does	Wrapping module · capability

file_open · file_read · file_write · file_seek · file_close	open, read, write, move the position of, and close a file	`files` · `cap file_system`
dir_make · dir_read · dir_close	make a directory, read its entries, close it	`files` · `cap file_system`
path_remove · path_rename · link_type	remove or rename a path, and ask what the path itself is without following symlinks	`files` · `cap file_system`
net_listen · net_accept · net_connect · net_port	listening socket · accept · connect to an address and port · ask the port	`net` · `cap net`
net_resolve	a name to an IPv4 address — DNS. A **separate** leaf from connecting, so a program that already has an address never touches name resolution	`net` · `cap net`
net_send · net_recv · net_close · net_pair	send · receive · close · a connected pair	`net` · `cap net`
reactor_new · r_read · r_write	make a reactor · read through a reactor · write through a reactor	no module — `cap io`
env_get	read an environment variable	no module — `cap env`
rng_next	next random number from a state (splitmix64)	`random` — no capability (pure)
hash_bytes · crc32	FNV-1a 64 hash · CRC-32 checksum	`hash` — no capability
sha256 · sha384 · sha512	SHA-256 · SHA-384 · SHA-512 — SHA-384 is SHA-512 with a different start value, keeping the first 48 bytes	`hash·hmac·tls13 · ed25519` — no capability
aes_ctr · ghash	AES-128-CTR keystream · GHASH accumulation — key, counter and accumulator are all 16 bytes. The counter and the accumulator are updated **in place**, so successive calls continue where the last one stopped	`gcm` — no capability
chacha_poly	one round of ChaCha20-Poly1305 — keystream and accumulation in **one pass**. Its meaning is exactly `chacha20` followed by `poly1305` over the	`aead` — no capability

	ciphertext it just made. The two computations use different execution resources (vector vs the integer multiplier), so interleaving overlaps them (measured 1,080 → 1,199 MB/s)	
`aes_gcm`	one round of AES-128-GCM — keystream and accumulation in **one pass**. Its meaning is exactly `aes_ctr` followed by `ghash` over the ciphertext it just made; only the speed differs (measured 2848 → 3661 MB/s). Sealing only — opening must attest the tag before it unseals, so it stays two passes	`gcm` — no capability
`chacha20 · poly1305`	ChaCha20 keystream · Poly1305 accumulation — key 32, counter block 16 (its first four bytes are the block number, little-endian), and the counter moves **in place**. `poly1305` accumulates into the state (five limbs of h, five of r) and zero-pads a tail that is not a multiple of sixteen. The width is chosen by the build (`lowentc --hw sse2 · avx2`)	`chacha · poly · aead` — no capability
`aes_round · aes_round_last`	one AES round — the 16-byte state in place. When the machine has the instruction the processor lowers to it (`lowentc --hw aes`); when it does not, to the same answer computed from tables	`aes` — no capability
`str_from_cstr`	scan a NUL-terminated C string into a `str`	C boundary — the VM says it cannot, with `E-VM-CSTR`

Table 32.6 — Leaf builtin ops and the modules that wrap them

Computation leaves stand only after `call_builtin` — `call_builtin sha256 msg out`. The name lives in that position and never becomes a global word: a word a program uses once should not cost every reader a name to remember. The stage names inside `pipe` and the type slot of `cast u8 x`

already work this way. Leaves that touch the operating system (`file_open`, `net_send`, ...) are called plainly — their first operand is a capability, so their specialness already shows.

The qualification for a leaf is one question: "can it be written in Lowent?" `rng_next` and the three hashes are pure computation yet leaves, because the processor fixes their algorithms so that the VM and the native build give bit-identical answers. Every leaf that needs a capability takes it as the **first operand** (chapter 16).

32.7 Common mistakes

Counter-example. Chaining writes that return `option` without asking

examples/ch32/mistake_chainnone.low

```
module mistake_chainnone .
rem run: greet [0,0,0,0,0,0,0,0,0,0,0,0]
rem trap: greet [0,0,0,0,0,0]

use fmt .

proc greet input buf mut slice u8 . output u64 . effects none .
do
  rem ✘ trusts that the previous write succeeded and chains on with `some_value`
  let p u64 be some_value (fmt.put_str buf 0 "hello") .
  let q u64 be some_value (fmt.put_byte buf p 32) .
  return some_value (fmt.put_u64 buf q 42) .
end
```

Output

```
$ lowentc --run greet mistake_chainnone.low [0,0,0,0,0,0,0,0,0,0,0,0]
greet([104,101,108,108,111,32,52,50,0,0,0,0]) = 8
  arg0 (written) = [104,101,108,108,111,32,52,50,0,0,0,0]
$ lowentc --run greet mistake_chainnone.low [0,0,0,0,0,0]
== ir diagnostics (1) ==
0:0 E-VM-NONE: some_value of none (panic)
```

The write ops of `fmt` return the next position as an `option u64`. With a twelve-slot buffer all three succeed and return 8, but with six slots `put_u64` runs out of room, returns `none`, and `some_value` stops. Failure in L0 modules is a **value**, not a trap, so the receiver asks at every step.

examples/ch32/chainnone_fixed.low

```
module chainnone_fixed .
rem run: greet [0,0,0,0,0,0,0,0,0,0,0,0]
rem run: greet [0,0,0,0,0,0]

use fmt .

proc greet input buf mut slice u8 . output u64 . effects none .
do
  rem ask at every step --- when short, return 0 to report "could not write"
  let p option u64 be fmt.put_str buf 0 "hello" .
  guard is_some p . else return 0 .
```

```
  let q option u64 be fmt.put_byte buf (some_value p) 32 .
  guard is_some q . else return 0 .
  let r option u64 be fmt.put_u64 buf (some_value q) 42 .
  guard is_some r . else return 0 .
  return some_value r .
end
```

Output

```
$ lowentc --run greet chainnone_fixed.low [0,0,0,0,0,0,0,0,0,0,0,0]
greet([104,101,108,108,111,32,52,50,0,0,0,0]) = 8
  arg0 (written) = [104,101,108,108,111,32,52,50,0,0,0,0]
$ lowentc --run greet chainnone_fixed.low [0,0,0,0,0,0]
greet([104,101,108,108,111,32]) = 0
  arg0 (written) = [104,101,108,108,111,32]
```

The fixed version returns 0 with the six-slot buffer. Notice that `hello` remains in the buffer. “All or nothing” is the promise of **a single op call**, not of a whole sequence of calls. If the whole sequence must be undone, treat only the positions that succeeded (`p`, `q`) as the valid length.

A common misconception. When the buffer is short, as much as fits is written

examples/ch32/all_or_nothing.low

```
module all_or_nothing .
rem run: try_put [0,0,0,0]

use fmt .

proc try_put input buf mut slice u8 . output u64 . effects none .
do
  rem six bytes into four slots --- instead of writing the first four, it writes none
  let p option u64 be fmt.put_str buf 0 "Lowent" .
  guard is_some p . else return 99 .
  return some_value p .
end
```

Output

```
$ lowentc --run try_put all_or_nothing.low [0,0,0,0]
try_put([0,0,0,0]) = 99
  arg0 (written) = [0,0,0,0]
```

C’s `snprintf` writes what fits and truncates. The L0 modules of the standard library are **all or nothing**. Asked to write six bytes into four slots, it writes no byte at all and returns `none`, leaving the buffer as `[0,0,0,0]`. A half-written buffer becomes silently wrong output, and the caller can rely on “none means the buffer is untouched” to continue down another path.

32.8 This chapter’s syntax at a glance

Shape	Meaning	Why

`use strings .`	import a standard module — the name is the file's `module` declaration	the module in `lib/str.low` is `strings`
`strings.starts_with line "GET "`	imported names are qualified by module	no glob imports
`fmt.put_str buf pos s → option u64`	write into the caller's buffer and return the next position	L0 never allocates — reentrancy is free
`none` **(not enough room)**	not a single byte was written	all or nothing — the promise of one call
`result t e`	a failure that says what failed	no traps, no silent truncation
`owned` **handles · pending bytes**	ownership that must be repaid	forgetting it is rejected at translation
`experimental · incubating · standard · deprecated`	maturity each module declares	a rung is the size of a promise, not a ranking

Table 32.7 — Shapes for using the standard library — shape · meaning · why it looks this way

Recap

The language takes only what cannot be expressed, leaves only thin pieces touching the outside, and the rest is library under the same rules. Libraries enter on the charter's evidence and write their maturity in the source. Modules divide into pure computation (L0), storage (L1) and host (L2) layers. Module names may differ from file names. Every module follows the caller's buffer, failure as a value, all or nothing, and ownership that must be settled.

33 Text and encodings — `strings`, `fmt`, `utf8`, `codec`, `hash`

In this chapter

33.1 Cut, find, assemble 421
33.2 UTF-8 — no replacement characters 422
33.3 Hex and base64 423
33.4 Hashes — three questions, three answers 424
33.5 Regular expressions — no backtracking 425
33.6 Common mistakes 425
33.7 This chapter's syntax at a glance 428

What to know first

chapter 9, Sequences · a string is not a type but `slice u8`
chapter 32, A map of the standard library · the caller's buffer, failure as a value, all or nothing

Looking back

How did chapter 9 answer the question “is there no separate string type”?

> **A.** A string literal is `slice u8`, and how many characters it has or whether it is valid UTF-8 is answered by library ops, not by the type. It is a choice to avoid the defects that arise the moment bytes and characters are treated as the same. This chapter tours that library.

The need for this chapter, and its context

Almost every program cuts, searches and joins strings, turns numbers into text, and moves bytes into hex or base64. In C, null-terminated strings and overflowing buffers follow all of these. Lowent’s text modules do not allocate, return results as views or into the caller’s buffer, and report failure as values. The shared conventions of chapter 32 show up here most often, so this is the first group of modules in the tour.

By the end of this chapter

You will learn to handle strings as views with `strings` and to assemble text into the caller’s buffer with `fmt`. You will see that `utf8` reports invalid bytes as `none` without replacement characters, and what `codec`’s hex and base64 and `hash`’s three kinds of hash (choosing slots, detecting damage, SHA-256) are each for. You will also see where the other text modules (`strbuf`, `utf16`, `unicode`, `regex`) fit.

The questions this chapter answers

1 Isn’t writing `guard is_some` every time you call `fmt.put_str` tedious?

33.1 Cut, find, assemble

examples/ch33/request.low

```
module request .
rem run: main

use strings .
use fmt .

proc main input out cap io . input al cap allocator . output u8 . effects io alloc .
do
  let line slice u8 be "GET /users/42 HTTP/1.1" .
  guard strings.starts_with line "GET " . else return 1 .
  let rest slice u8 be strings.remove_prefix line "GET " .
  let sp option u64 be strings.find rest " " 0 .
  guard is_some sp . else return 2 .
  let path slice u8 be subslice rest 0 (some_value sp) .
  let g option mut slice u8 be alloc_bytes al capacity 64 .
  guard is_some g . else return 3 .
  let buf mut slice u8 be some_value g .
  let p1 option u64 be fmt.put_str buf 0 "path=" .
  guard is_some p1 . else return 4 .
```

```
  let p2 option u64 be fmt.put_str buf (some_value p1) path .
  guard is_some p2 . else return 4 .
  let p3 option u64 be fmt.put_str buf (some_value p2) " len=" .
  guard is_some p3 . else return 4 .
  let p4 option u64 be fmt.put_u64 buf (some_value p3) (len path) .
  guard is_some p4 . else return 4 .
  let p5 option u64 be fmt.put_nl buf (some_value p4) .
  guard is_some p5 . else return 4 .
  let w u64 be write_out out 1 (subslice buf 0 (some_value p5)) .
  return 0 .
end
```

Output

```
$ lowentc --run main request.low
path=/users/42 len=9
main() = 0
```

- `strings.starts_with`, `strings.remove_prefix` and `strings.find` all take `slice u8`. What `remove_prefix` and `subslice` return is not a new buffer but a **view** pointing at the original bytes. There is no copy.
- `fmt.put_str buf pos s` and `fmt.put_u64 buf pos n` write into the caller's buffer `buf` from position `pos`, and give **the next position to write** as an `option`. If there is not enough room, not one character is written and it is `none`.
- When assembly is done, `subslice buf 0 end` is written to standard output.

The buffer was obtained with `cap allocator`, so `alloc` shows in the head. `strings` and `fmt` themselves do not allocate. Where the buffer comes from is the caller's business.

> **Q. Isn't writing `guard is_some` every time you call `fmt.put_str` tedious?**
>
> **A.** It is. In exchange, the moment room runs out it stops right there, and half-written output never goes out. To take a generous buffer and reduce repetition, use `strbuf`. `strbuf.append` appends to an owned buffer and gives failure as a `result`. If `strings` is reading (views), `strbuf` is writing.

33.2 UTF-8 — no replacement characters

examples/ch33/chars.low

```
module chars .
rem run: characters [236,149,136,235,133,149]
rem run: characters [97,98,99]
rem run: characters [236,149]
rem run: first_char [236,149,136,235,133,149]

use utf8 .

fn characters input s slice u8 . output u64 .
do
  let n option u64 be utf8.count_chars s .
  guard is_some n . else return 999 .
```

```
  return some_value n .
end

fn first_char input s slice u8 . output u64 .
do
  let c option u64 be utf8.decode s 0 .
  guard is_some c . else return 0 .
  return some_value c .
end
```

Output

```
$ lowentc --run characters chars.low [236,149,136,235,133,149]
characters([236,149,136,235,133,149]) = 2
  arg0 (written) = [236,149,136,235,133,149]
$ lowentc --run characters chars.low [97,98,99]
characters([97,98,99]) = 3
  arg0 (written) = [97,98,99]
$ lowentc --run characters chars.low [236,149]
characters([236,149]) = 999
  arg0 (written) = [236,149]
$ lowentc --run first_char chars.low [236,149,136,235,133,149]
first_char([236,149,136,235,133,149]) = 50504
  arg0 (written) = [236,149,136,235,133,149]
```

- `utf8.count_chars` counts characters. The Korean “안녕” is 6 bytes and 2 characters.
- The truncated `[236,149]` is not valid UTF-8, so it is `none`, and the example returns 999.
- `utf8.decode s 0` gives the one character at position 0 as a code point. 50504 is U+C548, “안”.

Many libraries insert a replacement character (U+FFFD) when they meet invalid bytes and carry on. Then the fact that the input was broken is buried in the output. `utf8` reports `none`. How to handle broken input is decided by the caller.

When exchanging values with the UTF-16 world (Windows APIs, Java, JavaScript), `utf16` joins and splits surrogate pairs over `slice u16`. Unmatched pairs are `none` too. Tables asking whether something is a letter, digit, space or zero-width are in the `unicode` module. Those tables were extracted mechanically from Unicode data, because hand-picked judgements silently give `false` instead of an error in ranges they missed.

33.3 Hex and base64

examples/ch33/hexout.low

```
module hexout .
rem run: main

use codec .

proc main input out cap io . input al cap allocator . output u8 . effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 16 .
  guard is_some g . else return 1 .
  let dst mut slice u8 be some_value g .
```

```
  let n option u64 be codec.hex_enc "Lowent" dst .
  guard is_some n . else return 2 .
  let w u64 be write_out out 1 (subslice dst 0 (some_value n)) .
  let nl u64 be write_out out 1 "\n" .
  let small option u64 be codec.hex_enc "too long for eight" (subslice dst 0 8) .
  guard not (is_some small) . else return 3 .
  return 0 .
end
```

Output

```
$ lowentc --run main hexout.low
4c6f77656e74
main() = 0
```

`codec.hex_enc src dst` writes `src` as hex characters into `dst` and gives **the number of bytes actually written**. The result is the first `n` bytes, not all of `dst`. The six characters “Lowent” become the twelve characters `4c6f77656e74`. Trying to encode 18 characters into 8 bytes of room gives `none` and writes nothing — all or nothing. `hex_dec`, `b64_enc` and `b64_dec` have the same shape.

> **A common misconception. Encoding is a kind of encryption**
>
> Hex and base64 are only ways of **transcribing** bytes as text; they keep no secrets. Anyone can reverse them. When secrecy is needed, use `aead`, which seals (chapter 36). `codec` and `aead` are different modules for the same reason as the charter’s “separate computation from authority” — different jobs do not get one name.

33.4 Hashes — three questions, three answers

examples/ch33/codes.low

```
module codes .
rem run: checksum [104,101,108,108,111]
rem run: slot [104,101,108,108,111] 16

use hash .

fn checksum input data slice u8 . output u64 .
do
  return hash.crc data .
end

fn slot input key slice u8 . input nslots u64 . output u64 .
  requires gt nslots 0 .
do
  return hash.bucket_of key nslots .
end
```

Output

```
$ lowentc --run checksum codes.low [104,101,108,108,111]
checksum([104,101,108,108,111]) = 907060870
  arg0 (written) = [104,101,108,108,111]
```

```
$ lowentc --run slot codes.low [104,101,108,108,111] 16
slot([104,101,108,108,111], 16) = 11
  arg0 (written) = [104,101,108,108,111]
```

The `hash` module gives different hashes to different questions.

Question	op	Hash used
Which slot of a hash map does it go in	`of` · `bucket_of` · `bucket_mask`	FNV-1a — fast and even
Was it damaged in storage or transit	`crc` · `crc_ok`	CRC-32 — catches accidental bit errors
Did someone change it on purpose	`digest` · `digest_ok`	SHA-256 — collisions are hard to find

Table 33.1 — Questions the `hash` module answers

`slot` puts the key "hello" into slot 11 of 16. `bucket_of` picks the slot with the remainder `mod`, so the result is always smaller than the slot count (chapter 4), and so the index bounds check of the hash table is removed. Using a slot-choosing hash to prevent tampering, or a cryptographic hash to choose hash map slots, are both the wrong tool. The names separate the questions.

33.5 Regular expressions — no backtracking

`regex` compiles a pattern once into a program (`slice u64`) and matches many inputs with the same program. The compiled result and the workspace are both the caller's slices, so there is no allocation.

Common regex engines **backtrack** — at a fork they go down one path to the end and come back when stuck — so for some patterns and inputs time grows exponentially in input length (ReDoS). `regex` uses the Pike VM approach, advancing all forks together one step at a time, so time is proportional to input length. In exchange, features that require backtracking, such as backreferences, are absent. What it does not do is written at the top of the module document.

33.6 Common mistakes

> **Counter-example. Trusting the result of removing a prefix without asking whether it was there**

examples/ch33/mistake_removeprefix.low

```
module mistake_removeprefix .
rem run: path_len [71,69,84,32,47,97]
rem run: path_len [80,79,83,84,32,47,97]

use strings .

rem ✘ trusts the result of removing without asking whether the prefix was there --- if a
bsent, the original comes back
fn path_len input line slice u8 . output u64 .
```

```
do
  let rest slice u8 be strings.remove_prefix line "GET " .
  return len rest .
end
```

Output

```
$ lowentc --run path_len mistake_removeprefix.low [71,69,84,32,47,97]
path_len([71,69,84,32,47,97]) = 2
  arg0 (written) = [71,69,84,32,47,97]
$ lowentc --run path_len mistake_removeprefix.low [80,79,83,84,32,47,97]
path_len([80,79,83,84,32,47,97]) = 7
  arg0 (written) = [80,79,83,84,32,47,97]
```

`strings.remove_prefix` returns **the original unchanged** when the prefix is absent. The design gives up reporting failure so that the result is always a usable view. So given `POST /a`, seven bytes come back as if “GET “ had been removed. Where the prefix must be present for the meaning to hold, ask with `starts_with` first.

examples/ch33/removeprefix_fixed.low

```
module removeprefix_fixed .
rem run: path_len [71,69,84,32,47,97]
rem run: path_len [80,79,83,84,32,47,97]

use strings .

rem ask first, and answer separately when it is absent
fn path_len input line slice u8 . output u64 .
do
  guard strings.starts_with line "GET " . else return 0 .
  return len (strings.remove_prefix line "GET ") .
end
```

Output

```
$ lowentc --run path_len removeprefix_fixed.low [71,69,84,32,47,97]
path_len([71,69,84,32,47,97]) = 2
  arg0 (written) = [71,69,84,32,47,97]
$ lowentc --run path_len removeprefix_fixed.low [80,79,83,84,32,47,97]
path_len([80,79,83,84,32,47,97]) = 0
  arg0 (written) = [80,79,83,84,32,47,97]
```

Counter-example. Treating the whole output buffer as the result of encoding

examples/ch33/mistake_wholedst.low

```
module mistake_wholedst .
rem run: sizes 0

use codec .

proc sizes input al cap allocator . output u64 . effects alloc .
do
```

```
  let g option mut slice u8 be alloc_bytes al capacity 16 .
  guard is_some g . else return 1 .
  let dst mut slice u8 be some_value g .
  let n option u64 be codec.hex_enc "Lowent" dst .
  guard is_some n . else return 2 .
  rem ✘ treating all of `dst` as the result adds four unwritten bytes --- 12 were writte
n, the buffer is 16
  return add (mul (len dst) 100) (some_value n) .
end
```

Output

```
$ lowentc --run sizes mistake_wholedst.low 0
sizes(0) = 1612
```

`codec.hex_enc` writes 12 bytes into the 16-byte buffer and returns 12 (the result 1612 puts the buffer size 16 and the count 12 side by side). Using all of `dst` as the result tacks four unwritten bytes onto the end. It is the same mistake as discarding the return value of C's `sprintf` and sending the whole buffer. The result is always `subslice dst 0 n`.

A common misconception. `len` is the number of characters, and a byte position is a character position

examples/ch33/bytes_not_chars.low

```
module bytes_not_chars .
rem run: lengths [236,149,136,235,133,149]
rem run: decode_at [236,149,136,235,133,149] 1
rem run: decode_at [236,149,136,235,133,149] 3

use utf8 .

rem `len` counts bytes --- "안녕" is 6 bytes, 2 characters
fn lengths input s slice u8 . output u64 .
do
  let chars option u64 be utf8.count_chars s .
  guard is_some chars . else return 0 .
  return add (mul (len s) 10) (some_value chars) .
end

rem no character starts at byte 1 --- the second character starts at byte 3
fn decode_at input s slice u8 . input at u64 . output u64 .
do
  let c option u64 be utf8.decode s at .
  guard is_some c . else return 0 .
  return some_value c .
end
```

Output

```
$ lowentc --run lengths bytes_not_chars.low [236,149,136,235,133,149]
lengths([236,149,136,235,133,149]) = 62
  arg0 (written) = [236,149,136,235,133,149]
```

```
$ lowentc --run decode_at bytes_not_chars.low [236,149,136,235,133,149] 1
decode_at([236,149,136,235,133,149], 1) = 0
  arg0 (written) = [236,149,136,235,133,149]
$ lowentc --run decode_at bytes_not_chars.low [236,149,136,235,133,149] 3
decode_at([236,149,136,235,133,149], 3) = 45397
  arg0 (written) = [236,149,136,235,133,149]
```

`len` counts bytes. "안녕" is 6 bytes and 2 characters, so `lengths` returns 62. Byte 1 lies in the middle of the first character, so `utf8.decode` returns `none` (0 here), and the second character "녕" (U+B155, 45397) starts at byte 3. To count or step through characters, use the ops of `utf8`; the number of columns a character takes on screen is yet another matter (chapter 37).

33.7 This chapter's syntax at a glance

Shape	Meaning	Why
`strings.starts_with s p` · `strings.find hay needle from`	ask · search (`option u64`)	on views — no copying
`strings.remove_prefix s p`	a view without the prefix — the original if absent	always a usable view, no failure
`fmt.put_str buf pos s` · `fmt.put_u64 buf pos n`	assemble into the caller's buffer — next position as `option`	all or nothing
`utf8.count_chars s` · `utf8.decode s at`	character count · code point at that position	invalid input gives `none`, not a replacement character
`codec.hex_enc src dst` · `b64_enc`	transcribe and return the count	the result is `subslice dst 0 n` — not encryption
`hash.bucket_of key n` · `hash.crc data` · `hash.digest`	choosing a slot · detecting damage · detecting tampering	a different hash per question
`regex`	compile once, match many inputs	no backtracking — time proportional to input length

Table 33.2 — Shapes of the text modules — shape · meaning · why it looks this way

Recap

`strings` gives copy-free views, `fmt` assembles into the caller's buffer and returns the next position as an `option`, and `strbuf` appends to an owned buffer. `utf8` and `utf16` report invalid input as `none` without replacement characters, and `unicode`'s tables were extracted mechanically. `codec` transcribes hex and base64 all or nothing. `hash` gives different hashes for slot choice, damage detection and tamper detection, and `regex` does not backtrack.

34 Containers and sorting — `sortlib`, `sortgen`, `hashmap`, `vecgen`, `spsc`

In this chapter

34.1 Sorting and finding `u64` 430
34.2 The type brings the criterion 430
34.3 A hash map over the caller's slice 431
34.4 A growing generic vector 432
34.5 A ring buffer passing values between flows 433
34.6 Other containers 434
34.7 Common mistakes 434
34.8 This chapter's syntax at a glance 437

What to know first

chapter 22, Generics · comparison is brought by the type, not a value
chapter 20, Allocators and fixed memory · allocators are handed over with `using`
chapter 32, A map of the standard library · buffers belong to the caller

Looking back

In chapter 22, C's `qsort` takes a comparison function as a value. What did it say Lowent's `sortgen` takes it by?

> **A.** The **type** brings the comparison. If the type being sorted satisfies the `ordered` trait and provides `less`, `sort_by` is monomorphised for that type and the comparison is built in as a direct call. This chapter tours that sort together with the standard library's containers.

The need for this chapter, and its context

Sorting, searching, hash maps and growing arrays go into almost every program. In other languages such containers usually allocate on the heap secretly and grow quietly when full. Lowent's containers come in two kinds: **fixed containers** whose backing slice the caller hands over (`hashmap`, `sortlib`), and **generic containers** that receive an allocator with `using` and grow themselves (`vecgen`, `mapgen`). Either way, where memory comes from is visible in the head.

By the end of this chapter

You will learn to sort and search `u64` slices with `sortlib.sort` and `searchlib.bsearch`, and to sort structs by a type-defined criterion with `sortgen.sort_by`. You will see how `hashmap` works over the caller's slice (empty slots and tombstones). You will watch `vecgen` receive an allocator and grow, with its effects depending on the allocator's type. You will also see `spsc`, which passes values between flows without locks, and where the other containers fit.

The questions this chapter answers

1 What happens if `less a a` is true?

34.1 Sorting and finding u64

examples/ch34/sorted.low

```
module sorted .
rem run: main

use sortlib .
use searchlib .

proc main input al cap allocator . output u8 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 40 .
  guard is_some g . else return 255 .
  var xs mut slice u64 be view_array u64 (some_value g) .
  set (index xs 0) 42 .
  set (index xs 1) 7 .
  set (index xs 2) 19 .
  set (index xs 3) 3 .
  set (index xs 4) 25 .
  sortlib.sort xs .
  let at option u64 be searchlib.bsearch xs 19 .
  guard is_some at . else return 254 .
  let missing option u64 be searchlib.bsearch xs 20 .
  guard not (is_some missing) . else return 253 .
  return narrow u8 (add (mul (index xs 0) 10) (some_value at)) .
end
```

Output

```
$ lowentc --run main sorted.low
main() = 32
```

- `alloc_bytes al capacity 40` gets 40 bytes, and `view_array u64` views them as a slice of five `u64` elements. Allocation happens only here.
- `sortlib.sort xs` sorts in place in ascending order. It is `effects none`.
- `searchlib.bsearch xs 19` gives the position of 19 in the sorted slice as an `option`, or `none` if absent. `lower_bound` gives the insertion position.

The smallest value 3 and 19's position 2 give 32. `bsearch` **trusts** that its input is sorted. Given an unsorted slice it gives a wrong answer. That trust is a precondition written in the module document.

34.2 The type brings the criterion

examples/ch34/rows.low

```
module rows .
rem run: main

use sortgen .

struct score do
  satisfies sortgen.ordered .
```

```
  points u64 .
  id u64 .
end

fn score.less input a score . input b score . output bool .
do
  return gt (field a points) (field b points) .
end

proc main input al cap allocator . output u8 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 48 .
  guard is_some g . else return 255 .
  var rs mut slice score be view_array score (some_value g) .
  set (index rs 0) (make score do points 70 . id 1 . end) .
  set (index rs 1) (make score do points 95 . id 2 . end) .
  set (index rs 2) (make score do points 80 . id 3 . end) .
  sortgen.sort_by score rs .
  return narrow u8 (add (mul (field (index rs 0) id) 100) (add (mul (field (index rs 1)
id) 10) (field (index rs 2) id))) .
end
```

Output

```
$ lowentc --run main rows.low
main() = 231
```

`score` declares that it knows order with `satisfies sortgen.ordered .`, and `score.less` gives the criterion “higher score first”. `sortgen.sort_by score rs` sorts by that criterion. The ids for scores 95, 80 and 70 are 2, 3 and 1 in order.

To sort descending or by several keys, write `less` that way. Instead of adding a mode argument, the type brings the meaning. `sort_by` is insertion sort, so it is **stable** (the order of equal scores does not change) and fast on nearly sorted input. For large arrays there is `sort_fast` (a generic quicksort). Scalars like `u64` cannot satisfy traits, so wrap them in a one-field struct. The layout stays 8 bytes.

> **Q. What happens if `less a a` is true?**
>
> **A.** Sorting may not stop or may give a wrong order. `less` must be a **strict weak ordering** — in particular nothing may come before itself. The mistake of writing `lt` where `le` belongs, or the reverse, happens here. A trait checks that an op **exists**, not that it keeps mathematical properties. Those properties are a contract written in the module document.

34.3 A hash map over the caller’s slice

examples/ch34/table.low

```
module table .
rem run: main

use hashmap .
```

```
proc main input al cap allocator . output u8 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 128 .
  guard is_some g . else return 255 .
  var slots mut slice u64 be view_array u64 (some_value g) .
  let a bool be hashmap.put slots 7 700 .
  let b bool be hashmap.put slots 9 900 .
  let c bool be hashmap.put slots 7 777 .
  let v option u64 be hashmap.lookup slots 7 .
  guard is_some v . else return 254 .
  let gone bool be hashmap.del slots 9 .
  let w option u64 be hashmap.lookup slots 9 .
  guard not (is_some w) . else return 253 .
  return narrow u8 (add (mul (div (some_value v) 100) 10) (hashmap.size slots)) .
end
```

Output

```
$ lowentc --run main table.low
main() = 78
```

`hashmap` is a `u64 → u64` open-addressing hash map. Its backing is a `mut slice u64` held by the caller, laid out as `[key0+1, value0, key1+1, value1, …]`. `size` is the slot count (`len / 2`). Initially everything must be 0 (empty), and `alloc_bytes` gives zero-filled bytes, so they are used as is.

- `put slots 7 700` inserts, and `put` again with the same key changes the value. When full it is `false`.
- `lookup` gives an `option`.
- `del` does not return a slot to empty; it covers it with a **tombstone**. Returning it to empty would break the probe chain continuing behind it, and later keys could not be found.

The price of this design is written in the document too. The two largest keys cannot be used, because they mark empty (0) and tombstone, and it does not grow automatically. Byte-string keys are handled by `strmap`, and a growing generic hash map by `mapgen`.

34.4 A growing generic vector

examples/ch34/growing.low

```
module growing .
rem run: main

use allocs .
use vecgen .

proc main input h cap heap . output u8 . effects heap state .
do
  var hb allocs.heap_bytes be spawn actor allocs.heap_bytes .
  let vo option (vecgen.vec u32 allocs.heap_bytes) using hb be vecgen.open u32 4 .
  guard is_some vo . else return 255 .
  var v vecgen.vec u32 allocs.heap_bytes be some_value vo .
  var i u32 be 0 .
  while lt i 100 . do
    let pushed bool be vecgen.append u32 allocs.heap_bytes v (mul i 2) .
```

```
    guard pushed . else return 254 .
    set i (add i 1) .
  end
  let x option u32 be vecgen.at u32 allocs.heap_bytes v 50 .
  guard is_some x . else return 253 .
  return narrow u8 (some_value x) .
end
```

Output

```
$ lowentc --run main growing.low
main() = 100
```

- `allocs.heap_bytes` is spawned and used as an allocator. Only `main`, holding `cap heap`, can spawn it (chapter 20).
- `vecgen.open u32 4` opens a vector with element type `u32` and initial capacity 4. The allocator is handed over with the binding's `using hb`.
- `vecgen.append … v x` asks the allocator for more and grows when out of room. On failure it is `false`. The vector already holds its own allocator, so `append` does not take `using`. Writing it is rejected with `E-ALLOC-USING-UNUSED`.
- `vecgen.at … v 50` gives element 50 as an `option`. 50 × 2 = 100.

The effect of `vecgen.append` is `state via a`. Here `a` is `heap_bytes`, so this instance's effect is `heap state`, and `heap` shows in `main`'s head too. Opening the same code with a bump over borrowed bytes shows only `state`. The allocator's type decides whether the container runs on a machine without an operating system.

> **A common misconception. A growing vector is hidden allocation after all**
>
> Allocation happens but is not hidden. `append`'s effects line carries the allocator's effect as is, and `heap` or `alloc` shows in the calling op's head. When the allocator runs short, `append` gives `false` — running out of memory is a value. Hidden allocation is allocation invisible in the head that stops the program when it fails.

34.5 A ring buffer passing values between flows

`spsc` is a lock-free single-producer, single-consumer ring buffer. One flow puts and another takes. The caller hands over the control and backing slots, and the ops that put and take receive `cap atomic` and move the cursors with atomic operations (chapter 27). It is used where interrupt handlers and ordinary code exchange values (chapter 30).

Lock-free data structures are notorious for being wrong only in rare orderings. `spsc`'s correctness was confirmed by **borrowing** the proof of an algorithm already proven in a weak-memory model. Multi-producer multi-consumer queues and seqlocks have no proof to borrow, so they were left out (chapter 47).

34.6 Other containers

Module	In one line
`strmap`	Byte string → `u64` hash map. Key bytes are kept in the caller's slice too
`mapgen`	Generic hash map `table k v`. Rehashes itself
`vecs · growvec`	Growing byte vectors. `growvec` is a short name for `vecgen.vec u8`
`nodelist`	Fixed-size intrusive list
`soa`	An experiment laying out an array of structs as per-field arrays

Table 34.1 — Other container modules

34.7 Common mistakes

Counter-example. Binary-searching a slice that was never sorted

examples/ch34/mistake_unsorted.low

```
module mistake_unsorted .
rem run: find42 0

use searchlib .

proc find42 input al cap allocator . output u64 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 40 .
  guard is_some g . else return 255 .
  var xs mut slice u64 be view_array u64 (some_value g) .
  set (index xs 0) 42 .
  set (index xs 1) 7 .
  set (index xs 2) 19 .
  set (index xs 3) 3 .
  set (index xs 4) 25 .
  rem ✘ binary search without sorting --- 42 is there, yet it is not found
  let at option u64 be searchlib.bsearch xs 42 .
  guard is_some at . else return 99 .
  return some_value at .
end
```

Output

```
$ lowentc --run find42 mistake_unsorted.low 0
find42(0) = 99
```

42 sits in slot 0, yet `bsearch` returns 99 (not found). Binary search compares with the middle value and discards half, and if the slice is not sorted the answer may be in the discarded half. It neither stops nor warns, which makes it the hardest kind of bug to find. With luck it even succeeds (in the same slice it finds 19). A binary search must always be preceded by `sortlib.sort`, and where code that keeps the order is scattered, use `lower_bound` to find the insertion point and keep the slice sorted.

Counter-example. Writing `less` non-strictly, like `ge`

examples/ch34/mistake_lessge.low

```
module mistake_lessge .
rem run: order_strict 0
rem run: order_loose 0

use sortgen .

struct strict_score do
  satisfies sortgen.ordered .
  points u64 .
  id u64 .
end

struct loose_score do
  satisfies sortgen.ordered .
  points u64 .
  id u64 .
end

fn strict_score.less input a strict_score . input b strict_score . output bool .
do
  return gt (field a points) (field b points) .
end

rem ✘ "comes first" written with `ge` --- equal scores each come before the other
fn loose_score.less input a loose_score . input b loose_score . output bool .
do
  return ge (field a points) (field b points) .
end

proc order_strict input al cap allocator . output u64 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 48 .
  guard is_some g . else return 255 .
  var rs mut slice strict_score be view_array strict_score (some_value g) .
  set (index rs 0) (make strict_score do points 70 . id 1 . end) .
  set (index rs 1) (make strict_score do points 70 . id 2 . end) .
  set (index rs 2) (make strict_score do points 70 . id 3 . end) .
  sortgen.sort_by strict_score rs .
  return add (mul (field (index rs 0) id) 100) (add (mul (field (index rs 1) id) 10) (fi
eld (index rs 2) id)) .
end

proc order_loose input al cap allocator . output u64 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 48 .
  guard is_some g . else return 255 .
  var rs mut slice loose_score be view_array loose_score (some_value g) .
  set (index rs 0) (make loose_score do points 70 . id 1 . end) .
  set (index rs 1) (make loose_score do points 70 . id 2 . end) .
  set (index rs 2) (make loose_score do points 70 . id 3 . end) .
  sortgen.sort_by loose_score rs .
  return add (mul (field (index rs 0) id) 100) (add (mul (field (index rs 1) id) 10) (fi
```

```
eld (index rs 2) id)) .
end
```

Output

```
$ lowentc --run order_strict mistake_lessge.low 0
order_strict(0) = 123
$ lowentc --run order_loose mistake_lessge.low 0
order_loose(0) = 321
```

All three scores are 70. `strict_score`, written with `gt`, keeps the insertion order 1, 2, 3 and returns 123. `loose_score`, written with `ge`, treats equal scores as “coming first” too, swaps them, and returns 321. The stable sort’s promise that “equal items keep their order” is broken. With another sorting algorithm it might even never finish or produce a wrong order. The trait checks only that `less` **exists**, so the property “nothing comes before itself” is for the writer to keep.

Counter-example. Writing `using` again for a container that already carries its allocator

examples/ch34/mistake_usingappend.low

```
module mistake_usingappend .
rem expect: E-ALLOC-USING-UNUSED

use allocs .
use vecgen .

proc main input h cap heap . output u8 . effects heap state .
do
  var hb allocs.heap_bytes be spawn actor allocs.heap_bytes .
  let vo option (vecgen.vec u32 allocs.heap_bytes) using hb be vecgen.open u32 4 .
  guard is_some vo . else return 255 .
  var v vecgen.vec u32 allocs.heap_bytes be some_value vo .
  var i u32 be 0 .
  while lt i 100 . do
    rem ✘ `using` is written again although the vector already carries its allocator
    let pushed bool using hb be vecgen.append u32 allocs.heap_bytes v (mul i 2) .
    guard pushed . else return 254 .
    set i (add i 1) .
  end
  let x option u32 be vecgen.at u32 allocs.heap_bytes v 50 .
  guard is_some x . else return 253 .
  return narrow u8 (some_value x) .
end
```

Output

```
$ lowentc --check mistake_usingappend.low
16:27 E-ALLOC-USING-UNUSED: this binding says which allocator to use, but the call it in
itialises does not draw from one (no `using` clause on that op). An object that already
carries its allocator — like `vecgen.append` on a vector — does not take the caller's ch
oice (RFC-0112 D8(5))
```

`vecgen.open` takes the allocator and stores it inside the vector. The later `append` uses the allocator the vector carries, so the binding's `using hb` means nothing. A meaningless mark becomes false information — "this call carves from `hb`" — so it is rejected with `E-ALLOC-USING-UNUSED`. Write `using` only where the allocator is **first received** (`open`).

A common misconception. Any `u64` can be a key in `hashmap`

examples/ch34/hashmap_topkeys.low

```
module hashmap_topkeys .
rem run: main

use hashmap .

proc main input al cap allocator . output u8 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 160 .
  guard is_some g . else return 255 .
  var slots mut slice u64 be view_array u64 (some_value g) .
  rem the two largest keys are used to mark empty slots and tombstones, so they cannot b
e stored
  let a bool be hashmap.put slots 18446744073709551615 40 .
  let b bool be hashmap.put slots 18446744073709551614 50 .
  let c bool be hashmap.put slots 18446744073709551613 60 .
  var r u8 be 0 .
  if a . do set r (add r 1) . end
  if b . do set r (add r 2) . end
  if c . do set r (add r 4) . end
  return r .
end
```

Output

```
$ lowentc --run main hashmap_topkeys.low
main() = 4
```

The result 4 means only the third `put` succeeded. The two largest keys (`18446744073709551615`·`18446744073709551614`) are used to mark empty slots and tombstones, so they cannot be stored and `put` returns `false`. This cost is stated at the top of the module's documentation. If keys may use the full range, put them in a struct and use `mapgen`.

34.8 This chapter's syntax at a glance

Shape	Meaning	Why
`sortlib.sort xs` · `searchlib.bsearch xs k`	sort `u64` in place · search sorted input (`option`)	no allocation — search trusts the order
`struct score do satisfies sortgen.ordered . … end` + `fn score.less`	the type brings the sort order	a type instead of a mode argument — `less` must be strict

`sortgen.sort_by score rs · sort_fast`	stable insertion sort · quicksort for large arrays	the choice is in the name
`hashmap.put slots k v · lookup · del`	a `u64 → u64` map on the caller's slice	`false` when full — deletion leaves a tombstone
`let vo … using hb be vecgen.open u32 4 .`	open a growing vector with an allocator	`using` only where it is first received
`vecgen.append u32 allocs.heap_bytes v x`	push while growing — `false` on failure	effects follow the allocator type (`state via a`)
`spsc`	lock-free single-producer single-consumer ring buffer	atomic operations — a borrowed proof

Table 34.2 — Shapes of containers and sorting — shape · meaning · why it looks this way

Recap

`sortlib` and `searchlib` sort and search `u64` slices in place, and with `sortgen` a type satisfying `ordered` brings the criterion. `hashmap` and `strmap` are fixed containers over the caller's slice that delete with tombstones. `vecgen` and `mapgen` receive an allocator with `using` and grow, their effects following the allocator type. `spsc` passes values between flows with atomic operations, and its correctness was confirmed with a borrowed proof.

35 Storage and handles — `pool`, `shard`, `budget`, `wire`

In this chapter

35.1 Generational handles 439
35.2 Brands — no mixing pools 440
35.3 Non-overlapping pieces — `shard` 441
35.4 Bit budgets — `budget` 442
35.5 Bit fields — `wire` 443
35.6 Other storage modules 443
35.7 Common mistakes 444
35.8 This chapter's syntax at a glance 446

What to know first

chapter 10, Groupings · linked by number, an old number sees the wrong node
chapter 13, Named types · a `newtype` is a new type with the same representation
chapter 19, Ownership · an `owned` value moves when handed over
chapter 20, Allocators and fixed memory · a bump takes back only its last piece

Looking back

In chapter 10, what inconvenience arose when linking a tree by numbers, and what did it say catches it?

> **A.** If a node is deleted and its number reused, whoever held the old number sees the wrong node. It said the standard library's `pool` catches this at run time with **generational handles** that carry a generation count alongside the number. This chapter covers that `pool` and other modules handling storage.

The need for this chapter, and its context

Bump allocators and regions are tools for "give it all back at once". They do not fit data made and deleted endlessly, like game objects, linked lists and cache pages. Using pointers for such data brings use-after-free, and using numbers brings number reuse. The modules in this chapter stop the same problem at different layers — run-time generation comparison, type brands, ownership tokens, translation-time bit budgets. Not a single new word was added. All are combinations of rules learned in earlier chapters.

By the end of this chapter

You will learn how `pool` lends fixed-size blocks through generational handles and recognises old handles after release. You will see how `newtype` brands stop the mistake of mixing different pools at translation. You will also see `shard` dividing storage into non-overlapping pieces with ownership tokens, `budget` checking a handle's bit budget at translation time, and `wire` handling bit fields with a single mask.

The questions this chapter answers

1 Generation comparison is a run-time check. Can't translation stop it?

35.1 Generational handles

examples/ch35/blocks.low

```
module blocks .
rem run: main

use pool .

newtype objects u8 .

proc main input al cap allocator . output u8 . effects alloc .
do
  let gm option mut slice u8 be alloc_bytes al capacity 64 .
  let gg option mut slice u8 be alloc_bytes al capacity 32 .
  guard is_some gm . else return 255 .
  guard is_some gg . else return 255 .
  var gens mut slice u64 be view_array u64 (some_value gg) .
  let po option (pool.block_pool objects) be pool.init objects (some_value gm) gens 16 .
```

```
  guard is_some po . else return 254 .
  var p pool.block_pool objects be some_value po .
  let ho option (pool.handle objects) be pool.take objects p .
  guard is_some ho . else return 253 .
  let h pool.handle objects be some_value ho .
  let before bool be pool.alive objects p h .
  let first bool be pool.release objects p h .
  let after bool be pool.alive objects p h .
  let again bool be pool.release objects p h .
  var code u8 be 0 .
  if before . do set code (add code 1) . end
  if first . do set code (add code 2) . end
  if after . do set code (add code 4) . end
  if again . do set code (add code 8) . end
  return code .
end
```

Output

```
$ lowentc --run main blocks.low
main() = 3
```

- `pool.init objects mem gens 16` divides the bytes `mem` handed over by the caller into 16-byte blocks and keeps a generation count per block in `gens`. The pool seals `mem` and `gens` inside, so later ops do not take them separately.
- `pool.take` lends one block and gives a **handle**. The handle holds the block number and the generation count at that moment.
- `pool.release` takes the block back and increments that block's generation count.
- An old handle after release has a different generation, so `alive` is false, and a second `release` is rejected too.

The result 3 means "alive before release (1), and the first release succeeded (2)", and that after release it was neither alive (4) nor did a second release succeed (8). Use-after-free and double free come out **as values** at run time.

> **Q. Generation comparison is a run-time check. Can't translation stop it?**
>
> **A.** Handles are values that can be copied, stored and written to files, so translation cannot know every handle that has gone stale. So this place is a dynamic check (in chapter 19's strength table, "dangling references of generational handles — dynamic"). Instead, the mistake translation can stop — mixing pools — is stopped at translation, as below. What translation stops and what run time stops is written at the top of the module document.

35.2 Brands — no mixing pools

Pools and handles carry a **brand** as a translation-time type parameter. One `newtype objects u8 .` declaration is one store.

examples/ch35/mixed.low

```
module mixed .
rem expect: E-TYPE-INSTANCE

use pool .

newtype meshes u8 .
newtype sounds u8 .

proc wrong input m mut pool.block_pool meshes . input h pool.handle sounds . output bool
  . effects none .
do
  return pool.release meshes m h .
end
```

Output

```
$ lowentc --check mixed.low
mixed.low:11:0 E-TYPE-INSTANCE: two instances of the same generic template are DIFFERENT
 concrete types — a container opened for one element type cannot be bound to a variable
declared for another. Monomorphization makes each instance a distinct struct (visible as
 `name#arg` in `--ir`), and their layouts differ; binding across them reads the same sto
rage at the wrong element width and yields a wrong value with no error at all. Declare t
he variable with the SAME instance, or convert explicitly (RFC-0021 · RFC-0084)
```

Trying to return a handle borrowed from the `sounds` pool to the `meshes` pool is rejected, because the two instances are different concrete types. A brand is a type, not a value, so the handle's fields do not grow. The cost is zero. Opening a store twice with the same brand is rejected too (`E-BRAND-REUSED`) — then one name would point at two, and the confusion the brand was meant to prevent would return.

35.3 Non-overlapping pieces — `shard`

`shard` divides one block of storage into non-overlapping pieces and gives a **token** per piece. A token is the right "this range is mine" and is an `owned` value.

examples/ch35/split.low

```
module split_tokens .
rem expect: E-OWN-MOVED

use shard .

newtype grid u8 .

fn halves_then_root output u64 .
do
  var r owned shard.token grid be shard.open grid 8 .
  var h shard.halves grid be shard.split_at grid r 4 .
  return shard.width grid r .
end
```

Output

```
$ lowentc --check split.low
split.low:12:0 E-OWN-MOVED: this `owned` value was already MOVED (consumed) — using it a
gain is use-after-move, which SPEC-004 §4.8 has always called a compile error and which
nothing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set <name> <new val
ue>` re-initialises the place — that is how a handle threads through a loop), or borrow
it LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered yet (E-IR-UNSU
P says so at the call site), so `f (ref h)` is not a way out today
```

`shard.split_at grid r 4` **consumes** the whole token `r` and splits it in two. Asking the width with `r` afterwards is `E-OWN-MOVED`. What stops this mistake is not the `shard` module but **the language**. The token is `owned`, so it leaves your hands the moment it is handed over. The idea of an access unit is not a new rule but the consequence of an existing one. Whether the pieces ride several flows or run sequentially, the fact that they do not overlap is itself the value.

The same shape is in `pagecache`. Pinning a page (`acquire`) consumes the cache token, so while the pin lives there is no token in hand to pass to eviction (`evict`). The defect of evicting a pinned page is stopped at translation.

35.4 Bit budgets — `budget`

To put a generational handle in one word you must decide “how many bits go to slot number, shard number and generation count”. `budget` writes those numbers in the source and has the tool keep them.

examples/ch35/budget_bad.low

```
module budget_bad .
rem expect: E-CONTRACT-IMPOSSIBLE

use budget .

fn too_wide output u64 .
do
  let ho option u64 be budget.pack 40 16 16 7 1 0 .
  guard is_some ho . else return 0 .
  return some_value ho .
end
```

Output

```
$ lowentc --check budget_bad.low
budget_bad.low:8:0 E-CONTRACT-IMPOSSIBLE: this call breaks the callee's `requires`, and
BOTH SIDES ARE CONSTANTS — so it can be decided here, now. It used to compile green and
trap at run time (E-VM-CONTRACT): a bit budget that does not fit (`slot + shard + genera
tion > word`) shipped as a runnable program. A contract that can be decided at compile t
ime IS decided at compile time (RFC-0104 §8-3)
```

`budget.pack 40 16 16 …` sums to 72 bits, which does not fit in a 64-bit word. All arguments are constants, so the contract violation is decided at translation (chapter 14). As the diagnostic’s story says, a program with a bit budget that did not fit was once shipped in runnable form. This module is the first place to use the check born of that. When a generation count fills up, the slot **retires** (`retired`). An old handle never comes back to life by secretly wrapping to 0.

35.5 Bit fields — `wire`

examples/ch35/fields.low

```
module fields .
rem run: mode_of 1234
rem run: with_mode 1234 5

use wire .

fn mode_of input w u64 . output u64 .
do
  return wire.pick 3840 w .
end

fn with_mode input w u64 . input m u64 .
  output u64 .
  requires le m 15 .
do
  return wire.merge 3840 w m .
end
```

Output

```
$ lowentc --run mode_of fields.low 1234
mode_of(1234) = 4
$ lowentc --run with_mode fields.low 1234 5
with_mode(1234, 5) = 1490
```

When several values sit side by side in one integer, `wire` handles a field with **a single mask**. The mask `0xF00` (3840) states both the position and width, “4 bits starting at bit 8”. `pick` extracts the field’s value, and `merge` swaps just that field. 1234 is `0x4D2`, so the field’s value is 4, and swapping in 5 gives `0x5D2` (1490). The two numbers “position” and “width”, which always diverged when written by hand, come from one mask and so cannot diverge. Packing named on/off settings into one word is the job of `flags`.

> **A common misconception. Modules like these must be written in unsafe code underneath**
>
> `pool`, `shard`, `budget` and `wire` are all written in the Lowent this book taught, without `unsafe`. Generation comparison is slice indexing and comparison, brands are `newtype` and type parameters, tokens are `owned`, and budgets are contracts. No new word or builtin was added. Because the library was written without privileges, the guarantees of programs using it are not broken.

35.6 Other storage modules

Module	In one line
`allocs`	Bump allocators, the allocator trait and the default allocator (chapter 20)
`segarena`	Grows in same-sized pieces and never moves what exists. Costs are written in op names

`pagecache`	Separates long-lived names (page numbers) from short-lived access rights (pins)
`segview`	Cursor, total length and flattening for views over scattered pieces
`lifemode` · `lifeatom`	Four ways to end a value — single ownership, thread-confined reference counting, atomic reference counting, external completion

Table 35.1 — Other storage modules

35.7 Common mistakes

Counter-example. Taking out the bytes of a returned block without asking

examples/ch35/mistake_staleuse.low

```
module mistake_staleuse .
rem trap: main

use pool .

newtype objects u8 .

proc main input al cap allocator . output u8 . effects alloc .
do
  let gm option mut slice u8 be alloc_bytes al capacity 64 .
  let gg option mut slice u8 be alloc_bytes al capacity 32 .
  guard is_some gm . else return 255 .
  guard is_some gg . else return 255 .
  var gens mut slice u64 be view_array u64 (some_value gg) .
  let po option (pool.block_pool objects) be pool.init objects (some_value gm) gens 16 .
  guard is_some po . else return 254 .
  var p pool.block_pool objects be some_value po .
  let ho option (pool.handle objects) be pool.take objects p .
  guard is_some ho . else return 253 .
  let h pool.handle objects be some_value ho .
  let done bool be pool.release objects p h .
  rem ✘ takes out the bytes of a returned block without asking --- the generation differ
s, so it is `none`
  var b mut slice u8 be some_value (pool.bytes objects p h) .
  set (index b 0) 7 .
  return index b 0 .
end
```

Output

```
$ lowentc --run main mistake_staleuse.low
== ir diagnostics (1) ==
0:0 E-VM-NONE: some_value of none (panic)
```

`pool.bytes` is **the only door** to a block's bytes, and it returns `none` when the generation does not match. In C, writing 7 into a freed block would silently corrupt another object; here the spot shows up as the value `none`. Taking it out with `some_value` without asking stops the program, as in this example. Stopping beats corruption, but code in which handles may live long should ask with `is_some` and handle stale handles separately.

Counter-example. Opening two stores under the same brand

examples/ch35/mistake_brandreuse.low

```
module mistake_brandreuse .
rem expect: E-BRAND-REUSED

use pool .

newtype objects u8 .

proc two_pools input al cap allocator . output u8 . effects alloc .
do
  let g1 option mut slice u8 be alloc_bytes al capacity 64 .
  let g2 option mut slice u8 be alloc_bytes al capacity 64 .
  let e1 option mut slice u8 be alloc_bytes al capacity 32 .
  let e2 option mut slice u8 be alloc_bytes al capacity 32 .
  guard is_some g1 . else return 255 .
  guard is_some g2 . else return 255 .
  guard is_some e1 . else return 255 .
  guard is_some e2 . else return 255 .
  var gens1 mut slice u64 be view_array u64 (some_value e1) .
  var gens2 mut slice u64 be view_array u64 (some_value e2) .
  let p1 option (pool.block_pool objects) be pool.init objects (some_value g1) gens1 16
.
  rem ✘ opens a second store under the same brand `objects` --- the types could no longe
r stop handles of the two pools mixing
  let p2 option (pool.block_pool objects) be pool.init objects (some_value g2) gens2 16
.
  return 0 .
end
```

Output

```
$ lowentc --check mistake_brandreuse.low
22:0 E-BRAND-REUSED: this brand already opened a storage somewhere else — a brand names
ONE storage instance, and a second creation site makes it name two. Then two different s
torages share a type, and handles from one type-check against the other: the very confus
ion the brand exists to refuse. Declare a second `newtype` and use it here (a brand carr
ies no data, so an extra one costs nothing at run time) (RFC-0104 §5.11 · §8-2)
```

The brand `objects` names **one** store. If the second `pool.init objects …` were accepted, two pools would share one type, and the types could no longer stop a handle from the first pool being returned to the second. Hence `E-BRAND-REUSED`. Declare one `newtype` per store; that one-line declaration is a cost-free distinction.

A common misconception. A value put into a `wire` field stops the program when it overflows

examples/ch35/wire_truncates.low

```
module wire_truncates .
rem run: put_mode 5
rem run: put_mode 20
rem run: fits_mode 20
```

```
use wire .

rem the field is four bits (0 … 15) --- 20 is cut down and 4 goes in
fn put_mode input m u64 . output u64 .
do
  return wire.put 3840 m .
end

rem if truncation is unwanted, ask first
fn fits_mode input m u64 . output bool .
do
  return wire.fits 3840 m .
end
```

Output

```
$ lowentc --run put_mode wire_truncates.low 5
put_mode(5) = 1280
$ lowentc --run put_mode wire_truncates.low 20
put_mode(20) = 1024
$ lowentc --run fits_mode wire_truncates.low 20
fits_mode(20) = 0
```

`wire.put` **truncates** values larger than the field. Putting 20 (`0b10100`) into a four-bit field keeps only the low four bits, 4, giving 1024 (`0x400`). The module documentation promises this and advises asking with `wire.fits` first if truncation is unwanted (20 gives 0, i.e. false). Bit fields going onto a wire often truncate by standard, which is why it was designed so. Where a stop is wanted, write `requires wire.fits …` as a contract.

35.8 This chapter's syntax at a glance

Shape	Meaning	Why
`newtype objects u8 .`	the brand of one store	cost-free distinction — mixing: `E-TYPE-INSTANCE`, opening twice: `E-BRAND-REUSED`
`pool.init objects mem gens 16`	a pool carving the caller's bytes into blocks (sealed in)	later ops do not take `mem`/`gens` again
`pool.take` · `pool.release` · `pool.alive`	borrow (generation handle) · return (bump generation) · ask	use-after-free and double free surface as values
`pool.bytes objects p h → option mut slice u8`	the only door to a block's bytes	`none` when the generation differs
`var r owned shard.token grid be shard.open grid 8 .`	an owned token for a non-overlapping piece	after splitting, the old token is `E-OWN-MOVED`
`budget.pack …`	check a bit budget by contract	a misfit is `E-CONTRACT-IMPOSSIBLE` at translation

`wire.pick mask w` · `wire.merge mask w v` · `wire.fits`	read · replace · check a bit field with one mask	position and width come from one number — `put` truncates

Table 35.2 — Shapes of stores and handles — shape · meaning · why it looks this way

Recap

`pool` lends blocks through generational handles and reports old handles after release and double frees as values at run time. `newtype` brands stop mixing pools at translation at no cost. `shard` and `pagecache` use `owned` tokens to stop overlap and eviction while pinned at translation. `budget` checks a handle's bit budget at translation time with contracts, and `wire` handles a bit field's position and width together with one mask. All are written with the language's rules and no new words.

36 Input/output, networking, time, randomness, cryptography

In this chapter

36.1 Buffered output — forget to flush and translation refuses 448
36.2 Networking — handles are resources 449
36.3 Randomness — reproducible sequences and operating-system entropy 450
36.4 Time — monotonic clocks and wall clocks 451
36.5 The HTTP request parser — its heart is rejection 452
36.6 Cryptography — the order of the stack and the missing top 453
36.7 Common mistakes 453
36.8 This chapter's syntax at a glance 456

What to know first

chapter 28, Input, output and files · a file handle is an owned value needing completion
chapter 16, Capabilities · `cap net`, `cap clock` and `cap random`
chapter 17, Designing failure · the heart of the HTTP parser is rejection

Looking back

In chapter 28, why did the answer of `files.read` split into three positions?

> **A.** Because if one value carries both "end" and "failure", a read failure is reported as "read everything". A line-counting program really did report a read failure as the success "0 lines". The modules in this chapter follow the same principle — one position per meaning, one kind per capability.

The need for this chapter, and its context

Modules touching the outside easily mix two things: **computation** and **authority**. Building text to output and emitting it to standard output, computing the next random value and obtaining entropy from the operating system, parsing a protocol's bytes and opening a connection — these are different jobs. Mixed, even pure computation drags capabilities and effects along and becomes hard to test. The modules in this chapter separate the two. We also see in what order the cryptographic modules are stacked towards TLS.

By the end of this chapter

You will gather output in a buffer with `outbuf` before emitting it and confirm that forgetting to flush is rejected at translation. You will learn to exchange bytes over `net`'s in-process connection pair, why deterministic `random.step` is separated from operating-system entropy `random.bytes`, and that monotonic clocks and wall clocks make different promises. You will also see what the HTTP request parser rejects, how the cryptographic modules are layered and what does not exist yet.

The questions this chapter answers

1 Why does the example use an in-process pair instead of opening a real TCP port?

36.1 Buffered output — forget to flush and translation refuses

examples/ch36/buffered.low

```
module buffered .
rem run: main

use outbuf .

proc main input out cap io . input al cap allocator . output u8 . effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 8 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  var p owned outbuf.pending be outbuf.buf_open 1 .
  let r1 result (owned outbuf.pending) outbuf.io_error be outbuf.buf_write out p buf "bu
ffered " .
  guard is_ok r1 . else return 2 .
  var p2 owned outbuf.pending be ok_value r1 .
  let r2 result (owned outbuf.pending) outbuf.io_error be outbuf.buf_write out p2 buf "o
utput\n" .
  guard is_ok r2 . else return 3 .
  var p3 owned outbuf.pending be ok_value r2 .
  let f result void outbuf.io_error be outbuf.buf_finish out p3 buf .
  guard is_ok f . else return 4 .
  return 0 .
end
```

Output

```
$ lowentc --run main buffered.low
buffered output
main() = 0
```

- `outbuf.buf_open 1` makes **pending output** to emit to standard output (1). It is `owned outbuf.pending`.
- `outbuf.buf_write out p buf s` gathers `s` into the buffer, emitting and continuing when the buffer fills. It takes a pending value and returns a new one (ownership moves along).
- `outbuf.buf_finish` emits the remaining bytes and ends the pending value.

The buffer is only 8 bytes, so it is flushed once partway through writing "buffered ". The output is the same. Gathering and emitting instead of `write_out` byte by byte is the value of buffering. Forget the final `buf_finish` and the bytes left in the buffer vanish. So translation refuses.

examples/ch36/unflushed.low

```
module unflushed .
rem expect: E-OWN-INCOMPLETE

use outbuf .

proc main input out cap io . input al cap allocator . output u8 . effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 64 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  var p owned outbuf.pending be outbuf.buf_open 1 .
  let r1 result (owned outbuf.pending) outbuf.io_error be outbuf.buf_write out p buf "lo
st?\n" .
  guard is_ok r1 . else return 2 .
  var p2 owned outbuf.pending be ok_value r1 .
  return 0 .
end
```

Output

```
$ lowentc --check unflushed.low
14:0 E-OWN-INCOMPLETE: this value is dropped automatically at the end of scope — but the
 program itself declares an op that takes this type `owned` BY VALUE and returns a `resu
lt`: finishing it CAN FAIL. An automatic drop is a RELEASE (total, non-suspending), and
it has nowhere to hand you that failure — it would SWALLOW it. A fallible finish (flush/
commit/close) is a COMPLETION and must be EXPLICIT: call it and handle the `result`. If
you really mean to discard the value and its failure, say so with `drop`. RFC-0058
```

`buf_finish` takes `owned pending` and returns a `result`, so `pending` is a type needing completion (chapter 19). The defect "the last line was not printed" shows before running.

36.2 Networking — handles are resources

examples/ch36/loopback.low

```
module loopback .
rem run: main
```

```
use net .

proc main input k cap net . input al cap allocator . output u8 . effects io alloc .
do
  let po option net.pair be net.pair_of k .
  guard is_some po . else return 1 .
  var p owned net.pair be some_value po .
  let sent option u64 be net.send_all k (net.pair_a p) "ping" .
  guard is_some sent . else return 2 .
  let g option mut slice u8 be alloc_bytes al capacity 16 .
  guard is_some g . else return 3 .
  let got option u64 be net.recv_once k (net.pair_b p) (some_value g) .
  let s result void net.net_error be net.shut_pair k p .
  guard is_some got . else return 4 .
  guard is_ok s . else return 5 .
  return narrow u8 (some_value got) .
end
```

Output

```
$ lowentc --run main loopback.low
main() = 4
```

`net.pair_of` makes a pair of connections joined to each other within the process. “ping” is sent through one side (`pair_a`) and received on the other (`pair_b`), getting 4 bytes. `net.shut_pair` closes the pair. Closing can fail and gives a `result`.

`net`’s connections and listeners are **resources** like `files`’ handles. If sockets are not closed, a server runs out of descriptors the longer it runs. `serve`, `dial` and `take` open TCP loopback connections, and every op takes `cap net` as its first argument. The head shows that code not using this module cannot reach the network.

> **Q. Why does the example use an in-process pair instead of opening a real TCP port?**
>
> **A.** Examples must run identically on the VM and natively and give the same answer. Real ports give results that depend on the machine’s state (ports already in use, firewalls). An in-process pair does not depend on outside state, so it is deterministic. Real server code uses the same `send_all` and `recv_once` through `serve` and `take`.

36.3 Randomness — reproducible sequences and operating-system entropy

examples/ch36/dice.low

```
module dice .
rem run: roll 42
rem run: roll 42
rem run: roll 43

use random .

fn roll input seed u64 . output u64 .
do
  let s1 u64 be random.step seed .
```

```
  return add (random.below_biased s1 6) 1 .
end
```

Output

```
$ lowentc --run roll dice.low 42
roll(42) = 3
$ lowentc --run roll dice.low 42
roll(42) = 3
$ lowentc --run roll dice.low 43
roll(43) = 5
```

`random.step` **computes** the next value from a seed. It is a pure `fn` with no capabilities or effects. Rolling twice with the same seed 42 gives 3 both times. Randomness that **must be reproducible**, as in simulations, tests and procedural generation, uses this.

Operating-system entropy is obtained with `random.bytes k dst`, which receives `cap random`. Randomness that **must not be predictable**, like keys and nonces, goes this way. The two jobs get different names and capabilities because merged, tests that must reproduce would depend on operating-system entropy, or conversely keys would come from a predictable sequence. `below_biased` is, as its name says, a biased range reduction. The name carries the meaning that it is not used where bias matters.

36.4 Time — monotonic clocks and wall clocks

examples/ch36/timing.low

```
module timing .

use clock .

proc elapsed_work input k cap clock . output u64 . effects none .
do
  let start u64 be clock.now_ns k .
  var i u64 be 0 .
  var acc u64 be 0 .
  while lt i 1000 . do
    set acc (wrap_add acc i) .
    set i (add i 1) .
  end
  return clock.since_ns k start .
end
```

Output

```
$ lowentc --check timing.low
== check: ok ==
```

`clock.now_ns` is a **monotonic** clock. It never goes backwards, so two readings can be subtracted (`since_ns`). It is not an absolute time. The wall clock (`local_packed`, `year_of` and so on) is the date and time people use, and it can go backwards with time zone changes or clock synchronisation.

Measuring elapsed time with a wall clock can give negative numbers. The two make different promises, so the ops differ.

Reading a clock gives different answers for the same input, breaking determinism. That is why `cap clock` is needed. But it leaves no trace outside, so the effect is `none` (chapter 16). This example's answer differs on each run, so this book's verification script only checks it. `sleep_ms` waits, so it is the `wait` effect.

36.5 The HTTP request parser — its heart is rejection

examples/ch36/request_line.low

```
module request_line .
rem run: classify [71,69,84,32,47,32,72,84,84,80,47,49,46,49,13,10,13,10]
rem run: classify [66,82,69,87,32,47,32,72,84,84,80,47,49,46,49,13,10,13,10]

use http .

fn classify input req slice u8 . output u64 .
do
  let m u64 be http.method_code req .
  guard http.version_ok req . else return 99 .
  return m .
end
```

Output

```
$ lowentc --run classify request_line.low [71,69,84,32,47,32,72,84,84,80,47,49,46,49,13,
10,13,10]
classify([71,69,84,32,47,32,72,84,84,80,47,49,46,49,13,10,13,10]) = 1
  arg0 (written) = [71,69,84,32,47,32,72,84,84,80,47,49,46,49,13,10,13,10]
$ lowentc --run classify request_line.low [66,82,69,87,32,47,32,72,84,84,80,47,49,46,49,
13,10,13,10]
classify([66,82,69,87,32,47,32,72,84,84,80,47,49,46,49,13,10,13,10]) = 0
  arg0 (written) = [66,82,69,87,32,47,32,72,84,84,80,47,49,46,49,13,10,13,10]
```

`http.method_code` gives a request's method as a number (`GET` is 1), and `http.version_ok` answers whether the request line's version is valid. The unknown method `BREW` is
0. The parser is pure computation and has no capabilities. Receiving bytes from a connection (`net`) and interpreting them (`http`) are different modules.

The design principle of this parser is **not to accept ambiguous input**. Request smuggling lives where servers and proxies accept an ambiguous request differently. Accepting leniently things like spaces in header names, duplicate length headers or malformed line endings creates that gap. This module does not build responses — it is the side that reads requests.

> **A common misconception. A lenient parser is kind to users**
>
> For configuration files people write, perhaps. In network protocols, one side's leniency becomes a difference of interpretation with the other, and that difference becomes attack surface. What is kind at a protocol boundary is saying the reason for a rejection by name.

36.6 Cryptography — the order of the stack and the missing top

The cryptographic modules are all pure computation (L0) and are stacked in layers towards TLS 1.3.

Layer	Modules
Derivation	`hash` (SHA-256) · `hmac` (HMAC-SHA256, HKDF)
Sealing (two suites)	`chacha`, `poly` → `aead` · `aes` → `gcm`
Key agreement	`x25519`
Signature verification	`bigint` → `rsa` (PSS) · `p256` (ECDSA) · `ed25519`
Signature generation	`p256` → `ecdsa` (derives the nonce without randomness)
Extracting keys and certificates	`pem` → `der` (a minimal parser extracting only public keys)
Protocol computation	`tls13` (key schedule, records, transcript, Finished)
Handshake	`tlssrv` (server handshake both ways + application data records — transport not yet)

Table 36.1 — The order the cryptographic modules are stacked in

The top of each module document warns **how it goes wrong when used alone**. `chacha` is not safe alone (→ `aead`), `aes` used alone is usually wrong (→ `gcm`), `poly`'s key must be fresh per message, and `gcm`'s nonce must never repeat. Filtering small-order points for `x25519` is the caller's job. If you use cryptography, sealing starts from `aead`.

And the top **does not exist yet**. `tlssrv` performs the whole handshake but transport over real sockets is not there yet, and `der` is not a certificate infrastructure (PKI); certificates are received from outside. The principle of not pretending to have what does not exist matters most in cryptography.

36.7 Common mistakes

> **Counter-example. Writing again with the old pending value passed to `outbuf.buf_write`**

examples/ch36/mistake_pendingmoved.low

```
module mistake_pendingmoved .
rem expect: E-OWN-MOVED

use outbuf .

proc main input out cap io . input al cap allocator . output u8 . effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 64 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  var p owned outbuf.pending be outbuf.buf_open 1 .
  let r1 result (owned outbuf.pending) outbuf.io_error be outbuf.buf_write out p buf "fi
rst\n" .
  guard is_ok r1 . else return 2 .
  rem ✘ writes again with the old pending value `p` that `write` took --- the new one is
 inside `r1`
```

```
  let r2 result (owned outbuf.pending) outbuf.io_error be outbuf.buf_write out p buf "se
cond\n" .
  drop r2 .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_pendingmoved.low
mistake_pendingmoved.low:15:0 E-OWN-MOVED: this `owned` value was already MOVED (consume
d) — using it again is use-after-move, which SPEC-004 §4.8 has always called a compile e
rror and which nothing enforced. To keep using it, either CONSUME AND PUT IT BACK (`set
<name> <new value>` re-initialises the place — that is how a handle threads through a lo
op), or borrow it LOCALLY with `ref h`. ☞ borrowing across an OP BOUNDARY is not lowered
 yet (E-IR-UNSUP says so at the call site), so `f (ref h)` is not a way out today
```

`outbuf.buf_write` takes an `owned pending` and returns a **new** pending value inside its `result`. The `p` you passed has already moved, so using it again is `E-OWN-MOVED`. Pending values move around so that exactly one value always knows what is left in the buffer; only then can translation count whether the final `buf_finish` was forgotten. Receive them in turn, `p` → `p2` → `p3`, as `buffered.low` in this chapter does.

Counter-example. Rolling twice from the same seed

examples/ch36/mistake_sameseed.low

```
module mistake_sameseed .
rem run: two_rolls 2

use random .

rem ✘ both rolls start from the same seed --- the two dice always match
fn two_rolls input seed u64 . output u64 .
do
  let a u64 be add (random.below_biased (random.step seed) 6) 1 .
  let b u64 be add (random.below_biased (random.step seed) 6) 1 .
  return add (mul a 10) b .
end
```

Output

```
$ lowentc --run two_rolls mistake_sameseed.low 2
two_rolls(2) = 55
```

`random.step` is a pure `fn`, so the same input always gives the same answer. Give both rolls the same `seed` and the two dice always match (55 with seed 2). There is no global random state, so passing on **the next state** is the caller's job.

examples/ch36/sameseed_fixed.low

```
module sameseed_fixed .
rem run: two_rolls 2
```

```
use random .

rem the next roll starts from the state the previous roll produced
fn two_rolls input seed u64 . output u64 .
do
  let s1 u64 be random.step seed .
  let s2 u64 be random.step s1 .
  let a u64 be add (random.below_biased s1 6) 1 .
  let b u64 be add (random.below_biased s2 6) 1 .
  return add (mul a 10) b .
end
```

Output

```
$ lowentc --run two_rolls sameseed_fixed.low 2
two_rolls(2) = 56
```

The fixed version starts the second step from the `s1` produced by the first and returns 56. That the same seed always yields the same two numbers is not a defect; it is this module's promise.

A common misconception. `recv_once` receives everything the other side sent in one go

examples/ch36/recv_partial.low

```
module recv_partial .
rem run: main

use net .

proc main input k cap net . input al cap allocator . output u8 . effects io alloc .
do
  let po option net.pair be net.pair_of k .
  guard is_some po . else return 1 .
  var p owned net.pair be some_value po .
  let sent option u64 be net.send_all k (net.pair_a p) "ping pong" .
  guard is_some sent . else return 2 .
  rem nine bytes were sent, but the receive buffer has four slots --- one receive takes
at most the buffer's size
  let g option mut slice u8 be alloc_bytes al capacity 4 .
  guard is_some g . else return 3 .
  let got option u64 be net.recv_once k (net.pair_b p) (some_value g) .
  let s result void net.net_error be net.shut_pair k p .
  drop s .
  guard is_some got . else return 4 .
  return narrow u8 (some_value got) .
end
```

Output

```
$ lowentc --run main recv_partial.low
main() = 4
```

Nine bytes, “ping pong”, were sent, but one receive into a four-slot buffer gives 4. A receive returns at most the buffer’s size, and only what has arrived by then. A stream has no message boundaries: what the sender sent in two parts may arrive at once, and what it sent at once may arrive in parts. If you need messages, prefix a length or define a delimiter, and receive repeatedly until the whole message is there.

36.8 This chapter’s syntax at a glance

Shape	Meaning	Why
`var p owned outbuf.pending be outbuf.buf_open 1 .`	pending output for standard output	forgetting it: `E-OWN-INCOMPLETE`
`outbuf.buf_write out p buf s → result (owned pending) …`	gather, flush when full, return a new pending value	one value knows — the old one is `E-OWN-MOVED`
`outbuf.buf_finish out p buf`	flush the rest and finish	completion — it can fail
`net.pair_of k · net.send_all · net.recv_once · net.shut_pair`	connected pair · send all · receive once · close	`cap net` first — a receive takes at most the buffer
`random.step seed · random.bytes k dst`	reproducible next state · OS entropy (`cap random`)	computation separated from authority
`clock.now_ns k · clock.since_ns k start`	monotonic clock — elapsed time	a different promise from wall time — `cap clock`, effect `none`
`http.method_code req · http.version_ok req`	parse the request line (pure)	ambiguous input is rejected
`aead · gcm · x25519 · ed25519 · tls13 · tlssrv`	sealing · key agreement · signatures · TLS computation	pieces unsafe on their own are flagged in their docs

Table 36.2 — Shapes of the I/O, network and crypto modules — shape · meaning · why it looks this way

Recap

`outbuf` gathers output before emitting it, and translation rejects pending values never flushed. `net` connections are resources opened with `cap net`, and closing can fail. `random.step` is reproducible pure computation, and `random.bytes` is entropy obtained with `cap random`. Monotonic clocks and wall clocks promise different things. The `http` parser is pure computation that rejects ambiguous input. The cryptographic modules are stacked as derivation, sealing, key agreement, signatures and TLS computation, and a TLS that transports does not exist yet.

37 The terminal — `term` and `tty`

In this chapter

37.1 `term` does not write to the screen 457
37.2 Redrawing only changed cells 458
37.3 The number of cells is not the number of bytes 459
37.4 Reading keys is computation 460

37.5 Raw mode is a capability 461
37.6 What is not built yet 462
37.7 Common mistakes 462
37.8 This chapter's syntax at a glance 464

What to know first

chapter 33, Text and encodings · `fmt` assembles into the caller's buffer and returns the next position
chapter 36, Input/output, networking, time, randomness, cryptography · separate computation from authority

Looking back

In chapter 36, why were the `http` parser and `net` different modules?

> **A.** Because receiving bytes (authority — needs `cap net`) and interpreting them (computation — pure) are different jobs. Merged, even pure interpretation drags capabilities and effects along and becomes hard to test. This chapter's terminal modules are split in half the same way.

The need for this chapter, and its context

Terminal programs do two things. They build control bytes to send to the screen, and they read the user's keys. For both, the part touching the operating system is very thin, and the rest is computation. How much of the screen to redraw, how many cells one Korean character takes, which key `ESC [ A` is — these are computation. The standard library split this computation into pure `term` and `tty` parsing, and the part touching the operating system into three builtin ops needing `cap tty`. As the last chapter of the library tour, it shows what this split actually buys.

By the end of this chapter

You will learn that `term` writes nothing to the screen but assembles ANSI control bytes into the caller's buffer, and how screen diffs redraw only changed cells. You will confirm that display width and grapheme clusters differ from byte counts. You will also interpret keys purely with `tty.parse_key`, and see why raw mode must always be restored when using the `cap tty` builtins that switch it on and off.

The questions this chapter answers

1 Doesn't cell width differ between terminals?

37.1 `term` does not write to the screen

Every op of `term` is `effects none`. It only assembles control bytes that move the cursor (`goto`), change colours (`sgr`, `color256`) or clear the screen (`clear`) into **the caller's buffer** and returns the next position to write. Actually emitting to the screen is done by `outbuf` (chapter 36).

Not knowing this leads to the dead end “the code runs but nothing appears on screen”. But thanks to this split, all screen code can be tested without a screen. Just compare the assembled bytes.

examples/ch37/assemble.low

```
module assemble .
rem run: paint [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

use term .

rem writes nothing to the screen; it appends control bytes to buf and returns how many b
ytes were written
proc paint input buf mut slice u8 . . output u64 . effects state .
do
  rem move the cursor to row 2, column 5 counted from 0; the terminal counts from 1, so
ESC [ 3 ; 6 H
  let p1 option u64 be term.goto buf 0 2 5 .
  guard is_some p1 . else return 0 .
  rem bold (1): ESC [ 1 m, continuing from the position the previous op returned
  let p2 option u64 be term.sgr buf (some_value p1) 1 .
  guard is_some p2 . else return 0 .
  let at u64 be some_value p2 .
  set (index buf at) 104 .
  set (index buf (add at 1)) 105 .
  rem reset attributes (0): ESC [ 0 m
  let p3 option u64 be term.sgr buf (add at 2) 0 .
  guard is_some p3 . else return 0 .
  return some_value p3 .
end
```

Output

```
$ lowentc --run paint assemble.low [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
paint([27,91,51,59,54,72,27,91,49,109,104,105,27,91,48,109,0,0,0,0,0,0,0,0]) = 16
  arg0 (written) = [27,91,51,59,54,72,27,91,49,109,104,105,27,91,48,109,0,0,0,0,0,0,0,0]
```

- `term.goto buf 0 2 5` writes the cursor-move bytes into `buf` from position 0 and returns the next position to write, 6, as an `option`. If the buffer is too short it writes nothing and gives `none`, because half-written control bytes would garble the screen.
- Rows and columns count **from 0**. The terminal convention (ANSI) counts from 1, so `goto` adds 1 and writes `ESC [ 3 ; 6 H` (27 91 51 59 54 72). The choice keeps the same base as array indexes.
- `term.sgr buf 6 1` writes `ESC [ 1 m`, which turns on bold, and the last `term.sgr … 0` writes `ESC [ 0 m`, which resets attributes. The 104 and 105 in between are the letters `hi`.
- Every op returns “the next position”, so the answer of one op becomes the `pos` of the next. Sixteen bytes are written in all, and the `buf` the VM shows is exactly those bytes.

37.2 Redrawing only changed cells

examples/ch37/diffing.low

```
module diffing .
rem run: main
```

```
use term .

proc main input al cap allocator . output u8 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 64 .
  guard is_some g . else return 255 .
  let out mut slice u8 be some_value g .
  let same option u64 be term.diff "hello   " "hello   " 8 out 0 .
  guard is_some same . else return 254 .
  let changed option u64 be term.diff "hello   " "help!   " 8 out 0 .
  guard is_some changed . else return 253 .
  return narrow u8 (add (mul (some_value same) 100) (some_value changed)) .
end
```

Output

```
$ lowentc --run main diffing.low
main() = 8
```

`term.diff prev next w out pos` compares the previous screen `prev` with the new screen `next` and writes into `out` only the changed runs, as “go there and write this”. If the two screens are the same it is 0 bytes — **0 bytes is exactly the answer “don’t touch the screen”**. When “hello” becomes “help!”, it is 8 bytes of cursor movement and two changed characters. Redrawing the whole screen every time flickers, and over remote connections transfers a lot.

37.3 The number of cells is not the number of bytes

examples/ch37/widths.low

```
module widths .
rem run: columns [104,105]
rem run: columns [236,149,136,235,133,149]
rem run: emoji_width 128512
rem run: clusters [101,204,129,120]

use term .

proc columns input row slice u8 . output u64 . effects none .
do
  let w option u64 be term.row_width row .
  guard is_some w . else return 999 .
  return some_value w .
end

proc emoji_width input cp u64 . output u64 . effects none .
do
  return term.cp_width cp .
end

proc clusters input row slice u8 . output u64 . effects none .
do
  let n option u64 be term.row_clusters row .
  guard is_some n . else return 999 .
```

```
  return some_value n .
end
```

Output

```
$ lowentc --run columns widths.low [104,105]
columns([104,105]) = 2
  arg0 (written) = [104,105]
$ lowentc --run columns widths.low [236,149,136,235,133,149]
columns([236,149,136,235,133,149]) = 4
  arg0 (written) = [236,149,136,235,133,149]
$ lowentc --run emoji_width widths.low 128512
emoji_width(128512) = 2
$ lowentc --run clusters widths.low [101,204,129,120]
clusters([101,204,129,120]) = 2
  arg0 (written) = [101,204,129,120]
```

- `hi` is 2 bytes and 2 cells.
- "안녕" is 6 bytes, but Korean characters are full-width, so it is 4 cells.
- The smiling face emoji (U+1F600) is 2 cells.
- The 4 bytes of `e` followed by a combining accent (U+0301) and then `x` are, to a person, two characters (é, x). `row_clusters` counts those clusters.

When aligning or cutting lines in a terminal, counting by bytes or code points misaligns cells for Korean and emoji. `row_width`, `fit_width` and `cluster_len` count by cells and clusters. The width table was extracted from Unicode data like the `unicode` module (chapter 33).

> **Q. Doesn't cell width differ between terminals?**
>
> **A.** It does. East Asian "ambiguous width" characters and new emoji are drawn differently by different terminals. `term` follows the Unicode standard's tables and writes that choice and its limits in the module document. Display width has a usable default — 1 even when wrong — so slight misalignment does not collapse the screen, but `unicode`'s tables judging whether something is a letter have no such default, so extracting them mechanically was essential.

37.4 Reading keys is computation

examples/ch37/keys.low

```
module keys .
rem run: first_key [27,91,65,113]
rem run: first_key [113]

use tty .

proc first_key input buf slice u8 . output u64 . effects none .
do
  let k option u64 be tty.parse_key buf 0 .
  guard is_some k . else return 0 .
  let code u64 be tty.key_of (some_value k) .
  if eq code (tty.key_up) . do
    return 1 .
```

```
  end
  if tty.is_char code . do
    return 2 .
  end
  return 3 .
end
```

Output

```
$ lowentc --run first_key keys.low [27,91,65,113]
first_key([27,91,65,113]) = 1
  arg0 (written) = [27,91,65,113]
$ lowentc --run first_key keys.low [113]
first_key([113]) = 2
  arg0 (written) = [113]
```

What a terminal sends for the up arrow is the three bytes `ESC [ A`. `tty.parse_key buf 0` reads those bytes as one value holding the key code and the length consumed. `tty.key_of` extracts the key code and `tty.len_of` the length. Ordinary characters are their byte value as is (1 … 255), and special keys are placed above 1000 (`key_up` is 1001). So one value distinguishes “this is a character” from “this is an arrow key”.

Reading a byte sequence as keys is computation, not input/output, so it is `effects none` and is verified by the VM/native cross-check without a screen. Unknown sequences or too few bytes are `none`.

37.5 Raw mode is a capability

The part touching the operating system is only three builtin ops: `tty_raw t on` (enter and leave raw mode), `tty_read t buf` (read key bytes) and `tty_size t` (screen size). All take `cap tty` as their first argument.

examples/ch37/rawmode.low

```
module rawmode .

use tty .

proc main input t cap tty . input al cap allocator . output u8 . effects alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 32 .
  guard is_some g . else return 1 .
  let buf mut slice u8 be some_value g .
  guard tty_raw t true . else return 1 .
  var going bool be true .
  var last u64 be 0 .
  while going . do
    let n option u64 be tty_read t buf .
    guard is_some n . else do
      set going false .
      continue .
    end
    let p option u64 be tty.parse_key (subslice buf 0 (some_value n)) 0 .
```

```
    if is_some p . do
      set last (tty.key_of (some_value p)) .
      if eq last 113 . do
        set going false .
      end
    end
  end
  let restored bool be tty_raw t false .
  return narrow_sat u8 last .
end
```

Output

```
$ lowentc --check rawmode.low
== check: ok ==
```

- The buffer is obtained **before** entering raw mode. If allocation failed after entering and it returned, the terminal would be left in raw mode.
- Bytes read with `tty_read` are interpreted with `tty.parse_key`, and pressing `q` ends it.
- At the end it always restores with `tty_raw t false`.

Raw mode **changes the user's terminal settings**. Even if the program dies, the change remains, and in the user's shell input becomes invisible and line breaks stop working. If such a capability floated around ambiently, any library could wreck someone's shell. So only its holder exercises it. This example waits for key input, so the verification script only checks it.

> **A common misconception. Restoring raw mode could just be enforced with ownership too**
>
> A good idea, but this edition does not do so. `tty_raw` is a builtin op returning a boolean, and there is no owned value representing "in raw mode". So forgetting to restore is not stopped at translation. It contrasts with `outbuf` and `files` enforcing completion through ownership, and for now it is a discipline a person must remember. The module document warns about this in bold too.

37.6 What is not built yet

The `tty` module document lists these itself — mouse reporting, bracketed paste, interpreting terminal query responses, and screen resize signal (SIGWINCH) integration. Each is a separate piece to build. Because the list is in the source and document, anyone building an editor with this module knows what they must do themselves before importing it.

37.7 Common mistakes

> **Counter-example. Creating a path that leaves raw mode without restoring it**

examples/ch37/mistake_rawleak.low

```
module mistake_rawleak .
```

```
use tty .

proc main input t cap tty . input al cap allocator . output u8 . effects alloc .
do
  guard tty_raw t true . else return 1 .
  rem ✘ allocates *after* entering raw mode, and leaves without restoring if that fails
  let g option mut slice u8 be alloc_bytes al capacity 32 .
  guard is_some g . else return 2 .
  let n option u64 be tty_read t (some_value g) .
  let restored bool be tty_raw t false .
  return 0 .
end
```

Output

```
$ lowentc --check mistake_rawleak.low
== check: ok ==
```

This code passes translation (restoring `tty_raw` is not enforced by ownership). But if allocation fails, it leaves through `return 2` with the terminal still in raw mode. The user's shell stops echoing input and line breaks go wrong. As `rawmode.low` in this chapter does, finish every preparation that can fail (getting the buffer) **before** entering raw mode, and make every path after entering it pass through `tty_raw t false`.

Counter-example. Cutting a row by byte count to fit the screen width

examples/ch37/mistake_bytecut.low

```
module mistake_bytecut .
rem run: cut_bytes [236,149,136,235,133,149]
rem run: cut_columns [236,149,136,235,133,149]

use term .

rem ✘ cuts at five bytes --- it splits the second character, leaving invalid text
proc cut_bytes input row slice u8 . output u64 . effects none .
  requires ge (len row) 5 .
do
  let w option u64 be term.row_width (subslice row 0 5) .
  guard is_some w . else return 999 .
  return some_value w .
end

rem ask where to cut by columns --- the longest prefix fitting three columns is the 3 by
tes of "안"
proc cut_columns input row slice u8 . output u64 . effects none .
do
  let n option u64 be term.fit_width row 3 .
  guard is_some n . else return 999 .
  return some_value n .
end
```

Output

```
$ lowentc --run cut_bytes mistake_bytecut.low [236,149,136,235,133,149]
cut_bytes([236,149,136,235,133,149]) = 999
  arg0 (written) = [236,149,136,235,133,149]
$ lowentc --run cut_columns mistake_bytecut.low [236,149,136,235,133,149]
cut_columns([236,149,136,235,133,149]) = 3
  arg0 (written) = [236,149,136,235,133,149]
```

Cutting “안녕” at five bytes splits the second character, which is no longer valid text, so `row_width` returns `none` (999). Sent to the screen as is, it would have printed broken characters. `term.fit_width row 3` gives, **as a byte length**, the longest prefix that fits in three columns — one “안”, 3 bytes. Always cut by that answer.

A common misconception. The bytes from one read always hold one whole key

examples/ch37/partial_key.low

```
module partial_key .
rem run: first_key [27,91]
rem run: first_key [27,91,65]

use tty .

rem there is no guarantee that one read holds a whole key
proc first_key input buf slice u8 . output u64 . effects none .
do
  let k option u64 be tty.parse_key buf 0 .
  guard is_some k . else return 0 .
  let code u64 be tty.key_of (some_value k) .
  if eq code (tty.key_up) . do
    return 1 .
  end
  return 3 .
end
```

Output

```
$ lowentc --run first_key partial_key.low [27,91]
first_key([27,91]) = 0
  arg0 (written) = [27,91]
$ lowentc --run first_key partial_key.low [27,91,65]
first_key([27,91,65]) = 1
  arg0 (written) = [27,91,65]
```

The up arrow is the three bytes `ESC [ A`, but a read may deliver only the first two. `tty.parse_key` then returns `none` (0 here), and once the third byte has arrived, it reads up (1). Do not throw the bytes away; prepend the leftover piece to the next read. Discarding `none` as “unknown key” becomes the bug where arrow keys occasionally vanish.

37.8 This chapter’s syntax at a glance

Shape	Meaning	Why

`term.goto` · `term.sgr` · `term.clear` (`buf pos …`)	assemble control bytes into the caller's buffer	nothing is written to the screen — test by bytes
`term.diff prev next w out pos`	bytes that redraw only the changed runs — 0 if equal	less flicker and less traffic
`term.row_width row` · `term.row_clusters row` · `term.cp_width cp`	columns · grapheme clusters · code point width	columns are not bytes
`term.fit_width row cols`	byte length of the longest prefix fitting the columns	never cut inside a character
`tty.parse_key buf at` · `tty.key_of` · `tty.len_of`	interpret key bytes (pure) — `none` when incomplete	reading keys is computation
`tty_raw t true` · `tty_read t buf` · `tty_size t`	raw mode · read key bytes · screen size	`cap tty` — always restore raw mode
`term.goto buf 0 2 5`	row and column counted from 0 — the bytes are `ESC [ 3 ; 6 H`	same base as array indexes; the op adapts to the terminal's 1-based count

Table 37.1 — Shapes of the terminal modules — shape · meaning · why it looks this way

Recap

`term` is a pure module assembling ANSI control bytes into the caller's buffer, and emitting to the screen is `outbuf`'s job. `diff` redraws only changed cells and is 0 bytes when nothing changed. Cell counts differ from byte counts, so count with `row_width` and `row_clusters`. `tty.parse_key` interprets key bytes purely. `tty_raw`, `tty_read` and `tty_size`, which touch the operating system, receive `cap tty`, and raw mode must always be restored.

Part X — Grounds: what has been proven

This part covers **how far** the claims this language makes about itself are true. “This is proven” came up several times in earlier chapters. This part honestly gathers what each of those claims means, what model it is about, and where it ends.

Why, and the tools (chapters 38 and 39) — tests show existence and proofs show absence. Why proof, exhaustive checking and cross-checking are overlapped, the scale of what is proven, and the six pieces of mathematics later chapters use (partial orders, lattices, fixed points, induction, abstract interpretation), from scratch.

Numbers and memory (chapters 40–43) — widening and division, the interval, relational and row-major proofs that remove bounds checks, the central theorem of ownership and borrowing, and loops that are safe however many times they run.

Effects and concurrency (chapters 44–47) — the theorem that effect declarations cover reality, the theorems that discipline removes races and parallel equals sequential, the weak memory model, and locks, where heavy tools are really needed.

Surface and identity (chapters 48 and 49) — the proof that whichever closer closes a form gives the same tree, and hashes that recognise the same program by meaning.

What is not proven (chapter 50) — the trusted base, the gap between model and implementation, and what remains by topic. That this part’s last chapter is the last chapter of the main text is no editorial accident. Quoting this part without reading that chapter makes this manual say things it did not say.

Theorems are put into plain sentences so you need not read Coq, and examples show which rejection by the compiler each theorem leads to. The mathematics needed is high-school level and is explained from scratch in chapter 39. Formal explanations written so the text continues if skipped are placed in maths boxes.

38 Why prove

In this chapter

38.1 Tests show existence, proofs show absence 468
38.2 Overlapping three layers 469
38.3 What is proven 470
38.4 Three ways of proving 471
38.5 Promises this part keeps 471

What to know first

chapter 31, Building and testing · tests, back-end cross-checks, contract cross-checks
chapter 14, Contracts · enforced contracts become facts that remove checks

Looking back

In chapter 31, what did checks like the back-end cross-check show about defects, and what could they **not** show?

> **A.** They show that defects **exist** but not that they **do not**, because two implementations giving the same answer are not thereby both right. And absence, it said, is the job of proofs. Part X covers those proofs.

The need for this chapter, and its context

Part X covers how far the claims this language makes about itself are true. “This is proven” came up several times in earlier chapters — widening, the index safety of `mod`, the borrowing rules, the determinism of parallelism. This part honestly gathers what each of those claims means, what model it is about, and where it ends. Its first chapter sets out why proofs are needed, and how three layers — proof, exhaustive checking and cross-checking — overlap.

By the end of this chapter

You will confirm with an example the distinction that tests show “exists (∃)” and proofs show “none (∀)”. You will learn what each of the three layers — proof, exhaustive checking, cross-checking — catches and misses, and the three strategies this repository’s proofs use (invariant preservation, reduction to an order structure, exhaustive checking). You will also understand the measured scale of what is proven, and the relation that proofs decide what the compiler must reject.

The questions this chapter answers

1 How do proofs connect to compiler code? Does a theorem change the compiler?

38.1 Tests show existence, proofs show absence

examples/ch38/average.low

```
module average .
rem run: mean2 10 20
rem trap: mean2 4294967295 1
rem test

fn mean2 input a u32 . input b u32 . output u32 .
do
  return div (add a b) 2 .
end

test small_numbers
do
  expect eq (mean2 10 20) 15 .
  expect eq (mean2 0 0) 0 .
  expect eq (mean2 1000000 3000000) 2000000 .
end
```

Output

```
$ lowentc --run mean2 average.low 10 20
mean2(10, 20) = 15
$ lowentc --run mean2 average.low 4294967295 1
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
$ lowentc --test average.low
  [PASS] small_numbers
== tests: 1 run, 1 passed, 0 FAILED ==
```

`mean2`, which computes the mean of two numbers, passes all three tests. But given 4294967295 and 1, `add a b` overflows `u32` and stops. Write a million more tests, and if the author does not pick large numbers, the defect does not show.

This is the nature of testing. A test can show "it is wrong on **this** input (∃)" but not "it is wrong on **no** input (∀)". Checking a million natural numbers is not checking all natural numbers. No amount of ∃ adds up to ∀. This gap is the one reason proofs are needed.

```
inputs of mean2: two u32 values = about 1.8 × 10^19 pairs

tests          (10,20) (0,0) (10^6,3·10^6)  three points tried → all right
counterexample (4294967295, 1)              a point never tried → overflows, stops
proof          ── every pair ──             speaks about all of them at once
```

In Lowent this defect is at least **not silent**. In C the overflowing sum would wrap and give a wrong mean. Here it stops (chapter 4). And the fixed version is written in a shape that cannot overflow.

examples/ch38/average_fixed.low

```
module average_fixed .
rem run: mean2 4294967295 1
rem ir
```

```
fn mean2 input a u32 . input b u32 . output u32 .
do
  let lo u32 be min a b .
  let hi u32 be max a b .
  return add lo (div (sub hi lo) 2) .
end
```

Output

```
$ lowentc --run mean2 average_fixed.low 4294967295 1
mean2(4294967295, 1) = 2147483648
$ lowentc --ir average_fixed.low
-- runtime checks (interval analysis: overflow · division · narrowing) --
   1 / 3 removed  (33%)
```

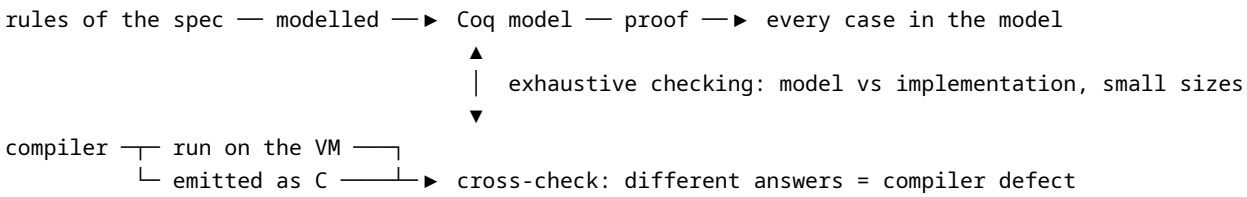
Adding half the difference of the two numbers to the smaller cannot overflow. `--ir` says it removed one of three checks. The remaining two stay even though they really cannot overflow, because the interval analysis does not know the relation between `min` and `max`. The analysis is wrong only **in the safe direction** — when it cannot prove, it keeps the check.

38.2 Overlapping three layers

Layer	What it does	How strong	What it misses
Proof	Every case, by logic	No counterexample can exist within the model	Powerless if the model is wrong
Exhaustive checking	Every case of a fixed size, actually run	Certain within that size	Cannot see larger inputs
Cross-checking	Two implementations on the same inputs	Catches disagreements	Misses when both are wrong the same way

Table 38.1 — Three layers of verification

They are overlapped because the three catch different kinds of mistakes. Drawn as who checks what against what:



Proofs look at the model, cross-checks look at the implementation, and exhaustive checking ties the two together. In this repository cross-checks caught defects the proofs missed, and proofs caught defects cross-checks missed. And this part does not mix three words. “Proven” means Coq checked every case, “exhaustively checked” means every case of a fixed size was run, and “sketch” means a person argued it but no machine checked.

> **A common misconception. A program written in a proven language is correct**
>
> Proofs are about **the language's rules**, not **your program**. The theorem that the borrowing rules are sound says only "programs passing translation have no borrowing violations"; it does not say the program gives the answer you want. `mean2` above has no borrowing violations and its overflow is not silent, but it was still a wrong design. Whether a program is correct is answered by contracts, tests and review.

38.3 What is proven

The proof files are all in `docs/proofs/coq/`. This is the scale counted for this edition.

Item	Count
Coq files	31 (including 1 checker extraction file)
Theorem · Corollary · Lemma	179 · 15 · 250
Example	33
`Qed`	481
`Admitted` · `Axiom`	0
Lines of proof script	10,350
Checkers	Rocq 9.2 (all) · Coq 8.20.1 (all but the two weak-memory files)

Table 38.2 — Scale of machine-checked proofs

In Coq, `Admitted` means "assume this is true and move on", and `Axiom` means "this is assumed". If even one such place exists, every theorem in the file relies on that assumption. Both are 0. What is proven and what is not yet is written in `docs/proofs/LEDGER.md`.

Grouped, it looks like this.

Group	In one sentence	Chapter
Tools	The six pieces of mathematics later chapters use — partial orders, lattices, fixed points, induction, abstract interpretation	chapter 39
Numbers	No widening silently changes a value, and the cases where division fails are known exactly	chapter 40
Bounds	Places where checks were removed are proven to be in range	chapter 41
Ownership and borrowing	Programs passing translation have no borrowing violations at run time	chapter 42
Loops	A loop that reaches a fixed point has no borrowing violations however many times it runs	chapter 43
Effects	Effect declarations cover the effects that actually happen, and optimisations are legitimate on top of that	chapter 44
Races and parallelism	Safe code has no data races, and parallel results are bit-for-bit identical to sequential	chapter 45

Weak memory	With the default memory ordering you may think sequentially, and weaker orderings add behaviours	chapter 46
Locks	A CAS spinlock gives mutual exclusion, and ascending lock order does not deadlock	chapter 47
Syntax	One statement is a one-statement block, and whichever closer closes it gives the same tree	chapter 48
Hashes	Comments do not change hashes, and equal encodings mean equal interfaces	chapter 49

Table 38.3 — Groups of what is proven, and the chapters of this part

The last chapter (chapter 50) gathers what all of these rely on and what they do not cover.

38.4 Three ways of proving

You do not need to read proof scripts. But knowing **how** something was proven gives a sense of how much to trust the theorem.

Invariant preservation. Show that "a property is kept every time the program takes a step". If it is true at the start (base) and a step carries truth to truth (preservation), it is true however many steps are taken. It is the program version of mathematical induction. The theorems for borrowing rules and loops take this approach.

Reduction to an order structure. Turn the problem into one about an order and answer it with properties of the order. Making "`u8` fits safely into `u32`" into a partial order turns "which type to pick when joining two types" into the well-known notion of a least upper bound. Numeric widening and the weak-memory model take this approach.

Exhaustive checking. Properties not yet proven for all sizes are run for every case within a fixed size. This is not proof, and this part says so.

38.5 Promises this part keeps

This part keeps three promises.

- **It does not mix words.** "Proven", "exhaustively checked" and "sketch" are different strengths. "This language is proven" is often really "a few short cases were run".
- **Every theorem comes with the defect it stops.** Theorems whose defect could not be named were left out. A theorem with no value to the reader has no reason to be here.
- **This part is not normative.** The language is defined by the specification's clause canon. If this part disagrees with it, this part is wrong.

And the ideas this language rests on, reduced to six lines, are these. All six are either backed by mathematics or written down as not backed — that distinction is the purpose of this part.

1. **Make wrong programs impossible to write.** Making them inexpressible is cheaper than catching them (chapter 42).
2. **Answer "I don't know" when you don't know.** Failure is a value — `option` or `result` — not a hidden exception.

3. **Costs must be visible.** Allocation, side effects and copies are all written in the source (chapter 44).
4. **When in doubt, keep the check.** Checks are removed only when there is a proof (chapter 41).
5. **Two implementations must give the same answer.** If the VM and native code differ, it is a compiler defect (chapter 31).
6. **Write down what was not done.** Assumptions, limits and what is unproven are recorded (chapter 50).

Q. How do proofs connect to compiler code? Does a theorem change the compiler?

A. Theorems decide **what the compiler must reject**. That the condition for `uN ⊑ iM` in the widening table is `N < M` (strict), and that blocking **only one** shape — a write borrow living across a loop meeting an owner write in the body — is enough, came from theorems. The other direction exists too. Turning a defect the implementation hit ("laundering a borrow through a function") into a theorem means the proof breaks the moment anyone removes that check. Stories are forgotten, but theorems remain (chapter 42).

```
theorem ──────────────────► decides which shapes the compiler must reject
                           e.g. the condition for uN ⊑ iM is N < M (strict)
a defect the compiler hit ► turned into a theorem
                           e.g. laundering a borrow through a function
                           remove that check and the proof breaks
```

Recap

Tests show that a wrong input **exists**, and proofs show that no input is wrong. Lowent overlaps proof, exhaustive checking and cross-checking, and does not mix "proven", "exhaustively checked" and "sketch". The 31 Coq files have no `Admitted` or `Axiom`. Proofs are built by invariant preservation, reduction to order structures and exhaustive checking, and they decide what the compiler must reject. Proofs are about the language's rules, not a guarantee that your program is correct.

39 The mathematical toolkit

In this chapter

39.1 Six tools 473
39.2 Sets and relations 474
39.3 Partial orders — incomparable pairs 474
39.4 Lattices and join — what to pick when combining 474
39.5 Monotone functions and fixed points — why analysis stops 475
39.6 Induction and invariants — dominoes 476
39.7 Abstract interpretation — ranges instead of values 477
39.8 Table of symbols 477
39.9 How the tools interlock 477

What to know first

chapter 38, Why prove · tests show existence (∃), proofs show absence (∀)
chapter 4, Numbers · width and sign fix a type, and widening happens automatically only when it keeps the value
chapter 7, Flow · `while` repeats its body while the condition holds

Looking back

What were the three ways of proving in chapter 38?

> **A.** Invariant preservation (a step carries truth to truth), reduction to an order structure (turn the problem into properties of an order) and exhaustive checking (run every case of a fixed size — not a proof). The first two need mathematical vocabulary. This chapter gathers that vocabulary from scratch.

The need for this chapter, and its context

The rest of Part X uses the words partial order, lattice, fixed point, induction and abstract interpretation without explanation. They look hard, but their meaning is mostly high-school mathematics — sets, inequalities, induction falling like dominoes. This chapter attaches the six tools to real Lowent programs once each. The symbols learned here are used unchanged in every later chapter. It is not a chapter that proves anything but one that prepares you to read.

By the end of this chapter

You will learn sets and relations, partial orders that allow incomparable pairs together with their three conditions, and the join and lattices used when combining two values. You will see that monotone functions and fixed points make loop analysis terminate, that induction and invariants show “however many steps”, and abstract interpretation, which computes with ranges instead of values, along with the direction of its soundness. You will also see a table of symbols and how the six tools interlock in later chapters.

The questions this chapter answers

1 What was proven in this chapter?

39.1 Six tools

Tool	In one line	Where it is used
Sets and relations	What exists, and what pairs with what	Everywhere
Partial orders	Comparison of size that allows incomparable pairs	chapters 40 and 46
Lattices and join	The smallest thing holding both when combining two	chapters 40 and 44

Monotone functions and fixed points	The point that no longer changes — loop analysis stops there	chapters 41 and 43
Induction and invariants	With a first piece and a rule for passing it on, every piece falls	chapters 42 and 43
Abstract interpretation	Compute with ranges of values instead of values	chapter 41

Table 39.1 — The mathematics later chapters use

39.2 Sets and relations

A **set** is a collection — `{0, 1, 2}`. A **relation** is a list of pairs. The relation “less than” is a collection of ordered pairs such as `{(0,1), (0,2), (1,2)}`. In talk about programs, a relation “widens safely” is placed over the set of types `{u8, u16, u32, u64, i8, …}`. That relation is written `⊑` and read “the left fits into the right without changing value”. It resembles `≤`, but it speaks not of numeric size but of **whether it can be held**.

39.3 Partial orders — incomparable pairs

Numbers are always comparable. Of 3 and 5, one is larger. Types are not. `u8 ⊑ u16 ⊑ u32 ⊑ u64` and `i8 ⊑ i16 ⊑ i32 ⊑ i64`, but neither `u32` nor `i32` holds all of the other. `u32` holds up to four billion and `i32` holds negatives. The structure that honestly captures the fact that neither can be called larger is a partial order.

Condition	Meaning	Example
Reflexivity	Always fits into itself	`u8 ⊑ u8`
Transitivity	Stepping stones connect	`u8 ⊑ u16` and `u16 ⊑ u32` give `u8 ⊑ u32`
Antisymmetry	Fitting both ways means the same	`A ⊑ B` and `B ⊑ A` give `A = B`

Table 39.2 — The three conditions of a partial order

These three must be **proven** for “safe widening” to mean anything. If transitivity broke, the language would allow `u8 → u16` and `u16 → u32` but block `u8 → u32`. Users could never find out why. The three are proven for the real types in chapter 40.

39.4 Lattices and join — what to pick when combining

When two values go into one operation, the processor must compute in **a type holding both**. There are several candidates, but the right one is the smallest. That “least common upper bound” is called the **join** and written `⊔`. A partial order where any two elements always have a join is a **lattice**.

examples/ch39/wider.low

```
module widening .
rem run: wider 200 60000
rem trap: wider 255 65535
```

```
rem adding u8 and u16 computes in the smallest type holding both (u16)
fn wider input a u8 . input b u16 . output u32 . do
  return add a b .
end
```

Output

```
$ lowentc --run wider wider.low 200 60000
wider(200, 60000) = 60200
$ lowentc --run wider wider.low 255 65535
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: integer overflow at the declared width (use wrap_*/sat_*, or prove th
e range)
```

`u8 ⊔ u16 = u16`. So even though the output type is `u32`, the addition happens in `u16`, and `255 + 65535` overflows `u16` and stops. The join is decided by the **operands**, not by the place that receives the result.

With mixed signs it defies intuition. `u8 ⊔ i8` is not `i8` but `i16` — holding both 255 from `u8` and −128 from `i8` needs a wider type. People get this wrong. Lowent does not pick this widening for you; it makes you write it.

examples/ch39/sign_join.low

```
module sign_join .
rem expect: E-TYPE-SIGN

rem the type holding both u8 and i8 is i16. The processor does not pick that widening fo
r you
fn mix input a u8 . input b i8 . output i16 . do
  return add a b .
end
```

Output

```
$ lowentc --check sign_join.low
sign_join.low:6:0 E-TYPE-SIGN: sign mismatch: no value-preserving widening exists (widen
 both to a strictly wider signed type, or bitcast_sign)
```

The “strictly wider signed type” in the diagnostic is exactly `u8 ⊔ i8 = i16`. Widening both to `i16` explicitly is accepted.

39.5 Monotone functions and fixed points — why analysis stops

A **monotone function** is one whose output does not get smaller as its input gets larger. A **fixed point** is a point where `f(x) = x` — applying once more changes nothing.

These two ideas do the work when the processor analyses a loop. The analyser does not know a variable’s range on entering the loop. So it starts narrow and widens repeatedly.

```
after round 1: i ∈ [0, 0]
after round 2: i ∈ [0, 1]
after round 3: i ∈ [0, 2]      … this way it never ends
```

So it uses **widening**. After seeing it widen a few times, it decides “this will keep growing” and pushes to the top at once. Combining `i ∈ [0, ∞)` with the loop condition `i < 10` gives `i ∈ [0, 9]`, and one more round still gives `[0, 9]`. **It has reached a fixed point** and the analysis ends.

examples/ch39/count_up.low

```
module count_up .
rem run: ten
rem proof

fn ten output u8 . do
  var i u8 be 0 .
  var s u8 be 0 .
  while lt i 10 . do
    set s (add s i) .
    set i (add i 1) .
  end
  return s .
end
```

Output

```
$ lowentc --run ten count_up.low
ten() = 45
$ lowentc --emit-proof count_up.low
# proven 1 certified 1
ten 14 add.i64 R-ARITH-RANGE 0 9 1 1 8 0
```

There is one proof line. `add i 1` adds within `i ∈ [0, 9]`, so it is proven not to exceed `u8` (`R-ARITH-RANGE 0 9`). `add s i` is not proven, because widening threw away the upper end of `s`. In reality `s` never exceeds 45, but the analysis does not know that. So the check stays — slow, but never wrong.

- That a fixed point always exists is the guarantee that the analysis **stops**.
- Widening goes **wide**. Going larger than reality only fails to remove checks; it never gives wrong answers. This direction is the heart of abstract interpretation.

39.6 Induction and invariants — dominoes

Mathematical induction goes like this. `P(0)` is true, and if `P(n)` is true then `P(n+1)` is true. Then `P(n)` is true for every `n`. In programs, `n` becomes “steps executed”, and `P` is called an **invariant** — a property true throughout execution.

> **The mathematics. The shape of the ownership and borrowing theorem**
>
> At the start the borrow state is clean (`P(0)`). Executing one event that passed the static check keeps a clean state clean (`P(n) → P(n+1)`). Therefore however many steps run, there is no borrowing violation (`∀n, P(n)`). The beauty of this argument is that **you need not know how many steps**. So the conclusion holds without knowing how many times a loop runs (chapters 42 and 43).

39.7 Abstract interpretation — ranges instead of values

To learn something about values without running the program, you must handle **sets of values** instead of single values. Handling sets as they are is too large, so a simple approximation is used. The typical one is the interval `[lo, hi]`.

```
if x ∈ [0, 5] and y ∈ [10, 20]
x + y ∈ [10, 25]      add the ends
x × y ∈ [0, 100]      with mixed signs, take min and max of four products
```

There is one rule. **The result of an abstract computation must include every value actually possible.** If `x` is always 3 but the analysis answers `[0, 5]`, it loses something but is safe (it cannot remove a check). If `x` can be 7 but the analysis answers `[0, 5]`, the moment a check is removed on that answer an out-of-range access happens. So every rule of the analysis leans “wide”.

> **A common misconception. The more precise the analysis, the better**
>
> **Direction** comes before precision. An analysis wrong on the wide side is only slow, but one wrong on the narrow side reads wrong memory. Each time a rule raising precision is added, the question is not “is it more precise” but “can it now be narrower than reality”. That is why this language keeps a light analysis that errs wide instead of a heavy one, and keeps checks where it falls short.

39.8 Table of symbols

Symbol	Read as	Meaning
`A ⊑ B`	A fits safely into B	The safe widening relation
`A ⊔ B`	The join of A and B	The smallest thing holding both
`[lo, hi]`	Interval	Values from lo to hi inclusive
`∀x, P(x)`	For all x, P	No exceptions
`∃x, P(x)`	There is an x with P	At least one
`P → Q`	If P then Q	Implication
`⊤`	Unknown	Any value possible
`A ∗ B`	A and B, on separate pieces	The separating conjunction (chapter 47)

Table 39.3 — Symbols in this part

39.9 How the tools interlock

```
partial order ⊑          defines "fits safely"               (numbers)
   │
   ├─ join ⊔             picks the type when combining       (numbers · effects)
   │
monotone functions       loop analysis climbs over them      (bounds)
on a lattice
   │
fixed point              where analysis stops                (bounds · loops)
   │
```

```
soundness of abstract     erring "wide" keeps conclusions safe  (bounds)
interpretation
   |
invariants and induction  show it holds throughout execution    (ownership · loops · races)
```

It reads bottom to top too. Induction keeps invariants, those invariants come from the soundness of abstract interpretation, abstract interpretation runs on a lattice, and a lattice stands on a partial order. Without the tools design becomes guesswork — a join table written by hand mixes in lines like `u8 ⊔ i8 = i8` and turns 255 into −1, loop analysis without widening never ends, and a borrow check without invariants becomes a pile of “allowed here, not there” rules.

Q. What was proven in this chapter?

A. Nothing. This chapter introduced vocabulary. That reflexivity, transitivity, antisymmetry, soundness of the join and existence of fixed points hold **for Lowent’s actual definitions** is the content from chapter 40 on. And the soundness of abstract interpretation must be checked rule by rule — “why is this one wide”. If even one rule is narrow, it leaks right there. That is why chapter 41 examines the rules one by one.

Recap

A partial order allows incomparable pairs and keeps reflexivity, transitivity and antisymmetry. The join is the smallest thing holding both and is decided by the operands (`u8 ⊔ u16 = u16`, `u8 ⊔ i8 = i16`). Loop analysis reaches a fixed point by widening and stops, and widening goes wide, keeping checks but never being wrong. Induction and invariants show “however many steps”, and abstract interpretation computes with ranges instead of values, always erring on the side of including reality.

40 Proofs about numbers — widening, narrowing, division

In this chapter

40.1 Widening is a partial order 479
40.2 Division and remainder 481
40.3 Theorems that `range` provides 482
40.4 Putting stopping into the model honestly 483
40.5 What is not proven 483

What to know first

chapter 4, Numbers · widening is automatic, narrowing is written
chapter 13, Named types · `range lo hi` is a contract that became the shape of a parameter
chapter 39, The mathematical toolkit · the three conditions of a partial order, and the join

Looking back

In chapter 4, `u8` and `i16` could be mixed but `u32` and `i32` could not. What separated them?

> **A.** Whether a value-preserving widening exists. Every value of `u8` (0 … 255) fits in `i16`, but large values of `u32` do not fit in `i32` of the same width. This chapter defines that "value-preserving widening" as a partial order from chapter 39 and reads the machine proof that it really is safe.

The need for this chapter, and its context

In C, `unsigned char small = 300;` silently becomes 44, and putting `-1` into an `unsigned int` gives 4294967295. Lowent allows only value-preserving widening automatically, and if losing is acceptable, makes you choose how to lose (`narrow_wrap`, `narrow_sat`, `narrow_try`). Saying that rule is right means a table of a few lines has no mistakes, and people get such tables wrong. So the definition was set up mathematically and the job of missing no case was given to a machine. These are the proofs that touch code most directly, so they come right after the tools.

By the end of this chapter

You will learn the four-line definition of the widening relation `⊑` and the theorem that it is a value-preserving partial order, along with properties of the join and narrowing. You will pick up exactly when division fails, the sign promise of `mod`, and the theorems about `range` that ground the removal of checks. You will also see how the three theorems of the type rules (`progress`, `preservation`, `values_fit`) put stopping into the model honestly.

The questions this chapter answers

1 Is it worth proving separately that widening cannot fail?

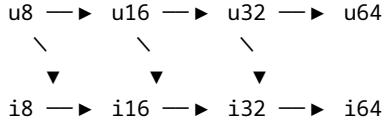
40.1 Widening is a partial order

`t ⊑ u` reads "any value of type `t` fits into type `u` without changing". The definition is four lines.

Left → right	Condition	Why
`uN` → `uM`	$N \leq M$	Between unsigned types only width must be larger
`iN` → `iM`	$N \leq M$	The same between signed types
`uN` → `iM`	$\mathbf{N < M}$	`u8` (255) does not fit in `i8` (127). Width must be **strictly** larger
`iN` → `uM`	none	There is nowhere to put negatives

Table 40.1 — The widening relation `⊑`

The $<$ in the third line is the point. Written as $\leq$, `u8 ⊑ i8` would be allowed and 255 would become −1. This one character decides value safety, and people get such things wrong.



```
→ is ⊑ (fits without changing the value); following arrows is ⊑ too (transitivity).
There is no same-width arrow down such as u8 → i8 or u16 → i16 (N < M),
and no arrow up from the i row to the u row (nowhere to hold a negative).
```

examples/ch40/order_strict.low

```
module order_strict .
rem expect: E-TYPE-SIGN

fn same_width input a u8 . output i8 .
do
  return a .
end
```

Output

```
$ lowentc --check order_strict.low
order_strict.low:6:0 E-TYPE-SIGN: sign mismatch: no value-preserving widening exists (wi
den both to a strictly wider signed type, or bitcast_sign)
```

examples/ch40/order_ok.low

```
module order_ok .
rem run: wider 200

fn wider input a u8 . output i16 .
do
  return a .
end
```

Output

```
$ lowentc --run wider order_ok.low 200
wider(200) = 200
```

What `NumericLattice.v` proves about this relation:

- **It preserves values** (`sub_preserves`). If `t ⊑ u` and `v` is in `t`'s range, it is in `u`'s range. This is why widening has no run-time check.
- **It is a partial order** (`sub_refl`, `sub_trans`, `sub_antisym`). If transitivity broke, the language would allow `u8 → u16` and `u16 → u32` but not `u8 → u32`. Managing the table by hand really does make such holes.
- **The join holds both and is one of the two** (`join_sound`, `join_is_an_operand`). When two values go into one operation, the type picked holds both values and does not invent a third type absent from the source (chapter 39's `u8 ⊔ u16 = u16`).
- **Narrowing has exactly one condition** (`narrow_ok_iff`). `narrow_try u8 300` never quietly succeeds giving 44.

> **The mathematics. The skeleton of the proof that widening preserves values**
>
> Split into four cases. If `uN ⊑ uM` ($N \le M$), both lower ends are 0 and the upper ends satisfy $2^N - 1 \le 2^M - 1$ — using only the fact that powers of 2 are monotonic. If `uN ⊑ iM` ($N < M$), then $N \le M - 1$, so $2^N - 1 \le 2^{M-1} - 1$. Signed pairs are the same, and `iN ⊑ uM` is false by definition, so there is nothing to do. The key lemma is just "if $a \le b$ then $2^a \le 2^b$". The whole proof stands on high-

> school exponent laws. What is hard is not the argument but **missing not a single case**, which is why it is left to a machine.

40.2 Division and remainder

Division is a place this language treats with unusual care.

examples/ch40/division.low

```
module division .
rem run: quot -7 2
rem trap: quot -128 -1
rem run: rest -7 3
rem run: rest 7 -3

fn quot input a i8 . input b i8 . output i8 . do
  return div a b .
end

rem the sign of mod follows the divisor
fn rest input a i8 . input b i8 . output i8 . do
  return mod a b .
end
```

Output

```
$ lowentc --run quot division.low -7 2
quot(-7, 2) = -3
$ lowentc --run rest division.low -7 3
rest(-7, 3) = 2
$ lowentc --run rest division.low 7 -3
rest(7, -3) = -2
$ lowentc --run quot division.low -128 -1
== ir diagnostics (1) ==
0:0 E-VM-OVERFLOW: division overflow (MIN / -1) at the declared width
```

Signed division `-7 / 2` truncates towards zero to −3. `-128 / -1` stops because `i8` has no 128. In C this one case is undefined behaviour. The sign of `mod` follows **the divisor** — `mod -7 3` is 2 and `mod 7 -3` is −2.

Theorem	Meaning
`div_unsigned_total`	Unsigned division always succeeds unless dividing by zero
`div_signed_failure_is_only_min_neg1`	The only failing signed division is `MIN / −1`
`mod_sign_follows_divisor`	The sign of `mod` follows the divisor
`mod_is_a_safe_index`	`mod h (len s)` is always at least 0 and less than `len s`

Table 40.2 — What is proven about division and remainder

```
unsigned div a b     b = 0                → stops
                     otherwise            → always a value
signed div a b       b = 0                → stops
                     a = MIN and b = −1   → stops (undefined behavior in C)
```

```
                    otherwise            → always a value (truncated toward 0)
mod h n             when n > 0           → 0 ≤ result < n  (a safe index)
```

The last theorem has great practical value.

examples/ch40/modslot.low

```
module modslot .
rem run: pick [5,6,7,8] 1000003
rem ir

fn pick input table slice u8 . input h u64 . output u8 .
  requires gt (len table) 0 .
do
  return index table (mod h (len table)) .
end
```

Output

```
$ lowentc --run pick modslot.low [5,6,7,8] 1000003
pick([5,6,7,8], 1000003) = 8
  arg0 (written) = [5,6,7,8]
$ lowentc --ir modslot.low
-- runtime checks (interval analysis: overflow · division · narrowing) --
   2 / 2 removed  (100%)
```

Hash tables always pick slots with `mod`, and because the result is proven in range, the index check disappears. What if `len s` is 0? It divides by zero and stops first. So on any path that yields a value, `len s > 0` is guaranteed. The argument has no gap.

40.3 Theorems that `range` provides

`range lo hi` written in a parameter's type position (chapter 13) makes the widening story one layer more precise.

Theorem	Meaning
`rsub_preserves`	Widening between ranges also preserves values
`radd_sound`	The sum of two ranges lies within the computed range
`radd_no_check`	If the result range fits in the type, no overflow check is needed
`idx_no_check`	If the index range lies within the length, no bounds check is needed
`rdisj_no_value`	Two disjoint ranges share no value (so one branch is dead)
`derive_minimal`	The derived range is the smallest (no wider than needed)

Table 40.3 — What is proven about ranges

`radd_no_check` and `idx_no_check` are what justify removing checks. The analysis that applies these two to real code is in chapter 41.

40.4 Putting stopping into the model honestly

The type rules are proven one layer deeper (`LowentType.v`). Over numeric expressions with width and sign, it adds binding, branches, comparison, narrowing, checked arithmetic and wrapping arithmetic, and proves three theorems.

- `progress` — a typed closed expression **is a value, stops, or takes a step**. It never gets stuck.
- `preservation` — taking a step does not change its type.
- `values_fit` — when execution reaches a value, that value is **always within its type's width**.

The third is special in this language. If a value over 8 bits lands in a `u8` place, the emitted C silently does something else. So ranges were put inside the type rules (attaching a type to a value requires range evidence) and proven preserved throughout execution.

And that the conclusion has **three** parts is the point. Checked arithmetic stops instead of producing a value on overflow. Put stopping into values and the theorem lies; leave stopping out and the theorem is false. So it is "value · stop · step". Preventing stops at translation time is the normal path's job, done by contracts and certificates (chapter 41).

```
a closed, typed expression e
   ├─► a value v      values_fit: v always fits the width of its type
   ├─► it stops       checked arithmetic overflowed · division by zero
   └─► one step → e'  preservation: e' has the same type (and again one of the three)
```

> **Q. Is it worth proving separately that widening cannot fail?**
>
> **A.** Yes (`widen_never_fails`). "Cannot fail" must be proven for widening sites to carry no runtime check. That is **why widening is free**. Conversely, `narrow_ok_iff` pins narrowing's success condition to "exactly within range", so `narrow_try u8 300` does not quietly give 44 but `none`. Failure is a value.

> **A common misconception. Since `values_fit` is proved, an overflowing value can never reach a `u8` position in this edition's compiler**
>
> The theorem is about the **type rules**. Whether the compiler follows those rules in every place is a separate matter. One example found while writing this manual: when a `pipe`'s `map` produces `u64` and `collect into` stores it in a `u8` buffer, this edition's tool silently wrapped the value for a long time (it is now rejected with `E-TYPE-COLLECT`, chapter 24). The gap between a proof that the rules are right and the fact that the implementation applies them everywhere shows up in places like this, and tests and the comparison of the two back ends fill it. Proofs decide what must be stopped; whether it was stopped is measured.

40.5 What is not proven

- **Floating point is not in this lattice.** Only integers were handled. `f32 → f64` preserves values, but the reverse and mixed operations are separate rules not proven here. Rounding, NaN and −0 are not covered either.
- **Widths are fixed at 8, 16, 32 and 64.** Arbitrary-width integers (such as `bits 7`) exist in the implementation (chapter 13) but lie outside this model, and `u128` exists in neither.

- **The theorems are about type rules.** Whether the compiler implements those rules exactly is confirmed by tests and back-end cross-checks (chapter 31).

Recap

Widening `⊑` is a partial order defined in four lines, and the condition for `uN ⊑ iM` is the strict `N < M`. Widening preserves values and cannot fail, the join is one of its two operands, and narrowing succeeds exactly within range. The only failing signed division is `MIN / −1`, `mod` follows the divisor's sign and is always a safe index. The `range` theorems ground check removal, and the type rules put stopping into the model honestly as "value · stop · step". Floating point is outside the proofs.

41 Proofs about bounds — intervals, relations, row-major addresses

In this chapter

41.1 Computing with intervals 485
41.2 Relations — what intervals cannot do 486
41.3 Four stages 487
41.4 The row-major proof 489
41.5 Trust proofs, but back trust with checks 490
41.6 What is not proven 491

What to know first

chapter 9, Sequences · writing the length condition as a contract removes index checks
chapter 14, Contracts · enforced contracts become facts that remove checks
chapter 39, The mathematical toolkit · abstract interpretation must err wide, and loop analysis stops at a fixed point
chapter 40, Proofs about numbers · `radd_no_check` · `idx_no_check`

Looking back

What were the two theorems that chapter 40 called the justification for removing checks, and what did each presuppose?

> **A.** `radd_no_check` says no overflow check is needed if the result interval fits the declared type, and `idx_no_check` says no bounds check is needed if the index is known to be at least 0 and less than the length. Both presuppose that **the fact is already known**. This chapter covers how the compiler obtains those facts — and what stops it from wrongly believing it has.

The need for this chapter, and its context

This language stops when integer arithmetic overflows and when an index goes out of range. So it would seem every operation carries a check and runs slowly. In practice most checks vanish at translation time, because not overflowing and being in range are proven. C does not check, so it is fast but reads other people's memory when out of range — half of all security incidents come from there. Lowent takes a third path: check, but remove the check on the spot when it is proven unnecessary. The claim "write honestly and it gets faster" rests entirely on this chapter's analysis, so we look at where it can go wrong and what backs it.

By the end of this chapter

You will confirm how interval analysis computes value ranges and that its soundness points one way. You will read, through `--emit-proof` output, how one-slot relations carry facts between variables that intervals cannot, and the four stages at which bounds checks vanish (direct · length stored in a local · capacity from a contract · row-major address). You will also see when the row-major proof collapses, the device by which the VM accuses the compiler when the analysis is wrong, the certificates left for each removed check and their checker, and the checks that remain even with contracts.

The questions this chapter answers

1 Why does a check remain for an index like `lru_idx`, where two values merge?

41.1 Computing with intervals

The compiler computes with intervals `[lo, hi]` instead of single values to decide overflow and bounds.

```
a ∈ [0, 100],  b ∈ [0, 100]
a + b ∈ [0, 200]        lo+lo, hi+hi
a − b ∈ [−100, 100]     lo−hi, hi−lo   subtraction flips --- a common place for mistakes
a × b ∈ [0, 10000]      with mixed signs, min and max of four products
```

examples/ch41/sum_contract.low

```
module sum_contract .
rem run: small_sum 100 100
rem ir

fn small_sum input a u8 . input b u8 . output u16 .
  requires le a 100 .
  requires le b 100 .
do
  return add (widen u16 a) (widen u16 b) .
end
```

Output

```
$ lowentc --run small_sum sum_contract.low 100 100
small_sum(100, 100) = 200
```

```
$ lowentc --ir sum_contract.low
-- runtime checks (interval analysis: overflow · division · narrowing) --
   2 / 2 removed  (100%)
```

The contract bounds both inputs to 100 or less, so the sum does not exceed 200 and does not overflow `u16`. Every check disappears. With the same shape but unbounded inputs, the checks remain.

examples/ch41/product_bare.low

```
module product_bare .
rem run: product 1000 1000
rem ir

fn product input a u64 . input b u64 . output u64 .
do
  return mul a b .
end
```

Output

```
$ lowentc --run product product_bare.low 1000 1000
product(1000, 1000) = 1000000
$ lowentc --ir product_bare.low
-- runtime checks (interval analysis: overflow · division · narrowing) --
   0 / 1 removed  (0%)
```

`1000 × 1000` does not overflow, but the analysis does not know how large `a` and `b` can get. So it keeps the check and confirms no overflow only when actual values arrive.

The theorems all point one way. **“Places said to be safe really are safe.”** The converse — safe but not said to be — is not guaranteed. That is only a performance loss.

```
analysis goes wide    →  checks remain     →  slow (safe)
analysis goes narrow  →  checks disappear  →  wrong memory access (dangerous)
```

So every time a rule is added, the question is the same — “can this rule be narrower than reality?”

41.2 Relations — what intervals cannot do

Intervals look at variables separately. Knowing `i ∈ [0, 100]` and `n ∈ [0, 100]` does not tell whether `i < n`. Yet that is exactly the fact array safety needs. So **one-slot relations** sit beside the intervals.

```
lenlt[i] = s     local i is less than len(local s)
lerel[i] = n     local i is less than local n
```

These facts come from branch conditions. Inside the body of `while lt i n`, `i < n` is true, and on the branch where the condition is false, `i ≥ n` is true. One slot is used instead of heavy relational domains (octagons, polyhedra) because of the shape of real code. Code walking an array is almost always shaped `while lt i (len s)`, and capturing the shape of real code exactly is worth more than sophisticated theory.

The rules for facts have three parts.

1. **Obtain.** Plant the fact on the branch where the condition is true.
2. **Carry.** Follow assignments — in `var x be j`, if `j < cap` then `x < cap`. At merges, keep a fact only if both paths agree. What holds on one side only is not a fact.
3. **Kill.** When a related variable changes, kill the fact in both directions. When a slice is rebound, every fact about its length dies.

The third is the core of safety. Keeping a fact alive too long is exactly wrong check removal, so killing is always the more aggressive side.

41.3 Four stages

Here are the shapes in which bounds checks vanish, in order of increasing strength. `--emit-proof` lists each check removed by proof on one line — op name, instruction position, operation, rule used, and the ranges the rule used.

examples/ch41/bound_stages.low

```
module bound_stages .
rem run: direct [1,2,3]
rem run: stored [1,2,3]
rem run: capacity [1,2,3,4] 3
rem run: grid [1,2,3,4] 2
rem proof

rem 1 --- the loop condition looks at len directly
fn direct input s slice u8 . output u64 . do
  var t u64 be 0 .
  var i u64 be 0 .
  while lt i (len s) . do
    set t (wrap_add t (widen u64 (index s i))) .
    set i (add i 1) .
  end
  return t .
end

rem 2 --- the length is stored in a local
fn stored input s slice u8 . output u64 . do
  let n u64 be len s .
  var t u64 be 0 .
  var i u64 be 0 .
  while lt i n . do
    set t (wrap_add t (widen u64 (index s i))) .
    set i (add i 1) .
  end
  return t .
end

rem 3 --- a contract states the capacity
fn capacity input s slice u8 . input cap u64 . output u64 .
  requires ge (len s) cap .
do
  var t u64 be 0 .
  var j u64 be 0 .
  while lt j cap . do
    set t (wrap_add t (widen u64 (index s j))) .
```

```
    set j (add j 1) .
  end
  return t .
end

rem 4 --- row-major address i·n + k
fn grid input a slice u8 . input n u64 . output u64 .
  requires le n 1000 .
  requires ge (len a) (mul n n) .
do
  var t u64 be 0 .
  var i u64 be 0 .
  while lt i n . do
    var k u64 be 0 .
    while lt k n . do
      set t (wrap_add t (widen u64 (index a (add (mul i n) k)))) .
      set k (add k 1) .
    end
    set i (add i 1) .
  end
  return t .
end
```

Output

```
$ lowentc --run direct bound_stages.low [1,2,3]
direct([1,2,3]) = 6
  arg0 (written) = [1,2,3]
$ lowentc --run stored bound_stages.low [1,2,3]
stored([1,2,3]) = 6
  arg0 (written) = [1,2,3]
$ lowentc --run capacity bound_stages.low [1,2,3,4] 3
capacity([1,2,3,4], 3) = 6
  arg0 (written) = [1,2,3,4]
$ lowentc --run grid bound_stages.low [1,2,3,4] 2
grid([1,2,3,4], 2) = 10
  arg0 (written) = [1,2,3,4]
$ lowentc --emit-proof bound_stages.low
# proven 12 certified 12
direct 12 index R-IDX-LENLT 0 281474976710655
direct 18 add.i64 R-ADD-LENLT 0 281474976710655 1 1 64 0
stored 14 index R-IDX-LENLT 0 281474976710655
stored 20 add.i64 R-ADD-LEREL 0 281474976710655 1 1 64 0
capacity 16 index R-IDX-LENLT 0 281474976710655
capacity 22 add.i64 R-ADD-LEREL 0 281474976710655 1 1 64 0
grid 8 mul.i64 R-MUL-CAP 0 1000 0 1000 64 0
grid 29 mul.i64 R-MUL-CAP 0 999 0 1000 64 0
grid 31 add.i64 R-ROW-CAP 0 999000 0 999 64 0
grid 32 index R-IDX-ROWMAJOR 0 999999
grid 38 add.i64 R-ADD-LEREL 0 999 1 1 64 0
grid 43 add.i64 R-ADD-LEREL 0 999 1 1 64 0
```

- **1 · Direct.** In `direct` the loop condition looks at `len s` directly. `index` was removed by `R-IDX-LENLT`.
- **2 · Length stored in a local.** `stored` puts the length in a local with `let n be len s`. It is a very common idiom. At one time the fact “this value is `len s`” was a property of a stack value that vanished the moment it was stored in a local, and this loop’s check stayed. Since the fact is

now carried by locals too, it is removed just like stage 1. This was a performance fix, not a safety fix — the check was there, so it was safe all along.

- **3 · Capacity from a contract.** In `capacity` the array size and loop bound are different variables. `j < cap` (loop condition) and `len s ≥ cap` (contract), so `j < len s`.
- **4 · Row-major address.** `grid` uses a **computed** index, `index a (add (mul i n) k)`. Neither intervals nor one-slot relations handle products, yet it was removed by `R-IDX-ROWMAJOR`. The product and sum of the address were proven along with it, by `R-MUL-CAP` and `R-ROW-CAP`.

The additions incrementing `i` and `j` vanished too, via `R-ADD-LENLT` and `R-ADD-LEREL`, thanks to the same relations. If `i < n` then `i + 1 ≤ n`, so it does not overflow.

41.4 The row-major proof

> **The mathematics. Proof that a row-major address is in range**
>
> There are three premises — the contract `len a ≥ p·q`, the outer loop condition `i < p`, and the inner loop condition `k < q`. The address `i·q + k` is computed with stopping multiplication and stopping addition. Then i ≤ p − 1, so i·q ≤ (p − 1)·q = p·q − q, and k ≤ q − 1, so i·q + k ≤ p·q − q + q − 1 = p·q − 1. Therefore address ≤ p·q − 1 < p·q ≤ len a. Only two inequalities were added.

High-school mathematics, but drop any premise and it collapses.

What was removed	Index check
Nothing	Disappears
The contract `len a ≥ n·n`	Remains
`i` bound by a different variable (`while lt i m`)	Remains
Address computed with `wrap_mul`	Remains

Table 41.1 — When the row-major rule holds

The last line is the subtlest.

examples/ch41/grid_wrap.low

```
module grid_wrap .
rem run: grid [1,2,3,4] 2
rem proof

rem the same address computed with a wrapping multiply
fn grid input a slice u8 . input n u64 . output u64 .
  requires le n 1000 .
  requires ge (len a) (mul n n) .
do
  var t u64 be 0 .
  var i u64 be 0 .
  while lt i n . do
    var k u64 be 0 .
    while lt k n . do
      set t (wrap_add t (widen u64 (index a (wrap_add (wrap_mul i n) k)))) .
      set k (add k 1) .
    end
    set i (add i 1) .
  end
```

```
  return t .
end
```

Output

```
$ lowentc --run grid grid_wrap.low [1,2,3,4] 2
grid([1,2,3,4], 2) = 10
  arg0 (written) = [1,2,3,4]
$ lowentc --emit-proof grid_wrap.low
# proven 5 certified 5
grid 8 mul.i64 R-MUL-CAP 0 1000 0 1000 64 0
grid 29 mul.i64 R-ARITH-RANGE 0 999 0 1000 64 0
grid 31 add.i64 R-ARITH-RANGE 0 999000 0 999 64 0
grid 38 add.i64 R-ADD-LEREL 0 999 1 1 64 0
grid 43 add.i64 R-ADD-LEREL 0 999 1 1 64 0
```

Same formula, but there is no `index` line. `wrap_mul` silently wraps on overflow. A wrapped value can get smaller, so the step $i{\cdot}q \le (p-1){\cdot}q$ breaks. In this example `n ≤ 1000` so it never actually wraps, but the rule holds only “for stopping multiplication”. **If the operation does not stop, it is not a fact.**

Then what if the contract’s product `p·q` itself overflows? The contract’s product is a stopping product, and contract checks are never removed, so if the body was reached it did not overflow. Where bounds checks were removed, the contract check is the only barrier. That is why contract checks are removed by neither modes nor optimisation.

In the development repository’s measurements, this rule cut the bounds checks of a matrix multiplication benchmark from 6 to 1 (the remaining one is `index c 0`, where `n ≥ 1` is missing from the contract), and those of an LRU benchmark whose contract states capacity from 10 to 4.

> **A common misconception. Fewer checks mean proportionally faster**
>
> The number of checks is not a proxy for cost. A check that never fires and is always branch-predicted correctly is practically free, and one benchmark cut checks from 7 to 0 with unchanged time. Checks are expensive where they block auto-vectorisation, as in matrix multiplication. So this part does not count fewer checks as an achievement. And arithmetic checks counted by `--ir` are recorded as proven while the C back end still emits the stopping calls — that number is **what is proven**, not **what disappeared**.

41.5 Trust proofs, but back trust with checks

Theorems say the rules are right, but whether the compiler applied them exactly must be confirmed separately. There are two devices.

Analysis self-accusation. The VM does not remove checks. It runs the places the compiler marked “may be removed”, and if a value is actually out of range, it accuses the compiler right there with `E-VM-ANALYSIS: … the interval/relational analysis is UNSOUND (this is a compiler bug)`. The native build has no checks and is fast, the VM has checks and reports when the mark is false, and the back-end cross-check runs both on the same inputs. This device has actually worked. While adding the row-major rule, `requires ge (len a) (mul n n)` was believed to be a checked

contract, but the processor could not read a contract of that shape, so there was really no check, and exactly this diagnostic appeared. **Trust only what is checked.**

Certificate recheck. The numbers at the end of each `--emit-proof` line are the grounds. For every removed check the compiler leaves arithmetic grounds (a Farkas certificate), and checkers written separately from the compiler recompute that arithmetic. The `certified` count on the first line is how many passed. One checker is **extracted** into OCaml from the Coq code that proved the rule's soundness (`LowentCert.v`, `LowentCertExtract.v`). Nobody confirms "did I transcribe this arithmetic correctly" for a hand-ported checker, but the extracted one is the very function the theorem talks about. Still, this only makes the rule's **arithmetic** trustworthy; "does that fact really hold at that place" is outside the checker's jurisdiction.

> **In practice. A tool that did not stop**
>
> While making interval analysis examples, an op with a loop running `n` times finished `--check` at once but `--ir` ran past 60 seconds without ending. `--ir` derives boundary values from the contract and actually runs the op, and with `n = 2^64 - 1` that loop would not finish within a human lifetime. That run was given a step budget, and cases hitting the budget are **counted as skipped**. Not passing them silently is the point. The moment a tool says it checked what it did not, it becomes a lie. The `--run` users invoke has no budget — only tests the tool makes for itself may be cut off.

> **Q. Why does a check remain for an index like `lru_idx`, where two values merge?**
>
> **A.** At a merge a fact survives only if both paths agree. If `lru_idx` arrives as `0` on one side and `j` on the other, the fact `j < cap` belongs to one path and dies at the merge. Even if `0 < cap` is known separately, one-slot relations do not gather and carry "both are less than cap". This analysis would rather err by keeping a check than by keeping a fact alive too long.

41.6 What is not proven

Some checks do not go away even with contracts. These are the places measured.

Remaining place	Why it cannot be removed
`index c 0` in matrix multiplication	`n ≥ 1` is missing from the contract (adding it closes it)
`index keys lru_idx` in LRU	It merges `0` and `j`, so the relational fact dies at the merge
`index s i` in the sieve (`while lt (mul i i) n`)	`i·i < n ⇒ i < n` is another nonlinear shape, rare in real code, so no rule was made
Partition indices in sorting	There is nowhere to carry `hi ≤ len s` non-strictly (relations are strict only for now)

Table 41.2 — Checks that remain, and why

- **Widening discards information.** It goes wide to reach a fixed point quickly, so checks remain even where safe (chapter 39's `count_up`). A loss, not a danger.

- **Facts about array contents have no place in this domain.** “Every element is less than 10” is closed by no `requires`. It is the tool’s limit, not the program’s defect, and since the check stays the answer is still right.
- **Floating-point intervals are not handled.**
- **The correspondence between analysis rules and theorems is empirical.** Self-accusation, certificate rechecks and back-end cross-checks back it (chapter 50).

Recap

Interval analysis computes with ranges instead of values; erring wide is only slow, erring narrow is dangerous. One-slot relations obtain facts like `i < len s` from branch conditions, carry them, and kill them aggressively when related variables change. Bounds checks vanish in four stages — direct · length stored in a local · capacity from a contract · row-major address — and `--emit-proof` lists each rule on one line. The row-major proof holds only for stopping multiplication, and contract checks are never removed. The VM’s self-accusation and a certificate checker extracted from Coq back trust, and checks remaining because of merges, nonlinearity or array contents are written down as remaining.

42 Proofs about ownership and borrowing

In this chapter

42.1 Two machines and three events 493
42.2 The central theorem 494
42.3 Shapes rejected and shapes accepted 495
42.4 Finding an implementation defect while widening the model 495
42.5 Between model and implementation 496
42.6 What is not proven 496

What to know first

chapter 12, Borrowing · many readers or one writer
chapter 19, Ownership · the strength of memory rules — static, dynamic, proven
chapter 39, The mathematical toolkit · invariants and induction — a step carries truth to truth

Looking back

chapter 12’s `stale.low` wrote 5 to `n` while `r` held a read borrow of `n`. What rejected it, and what does that rule protect?

A. It was rejected with `E-EXCL`. Borrows of one value must be many readers or one writer, and the rule protects **the reader’s belief** — that the value does not change while it is looking. This chapter covers whether programs the compiler “passes” under this rule really have no violations at run time.

The need for this chapter, and its context

The borrowing rules are the foundation of this language's memory safety and absence of data races. But a rule existing and a rule being **sufficient** are different. If the rule has a hole, the compiler's "pass" means nothing. This chapter's theorems give that "pass" meaning. And in the course of proving, one real compiler defect was turned into a theorem. It is the story that best shows what proofs do, so it comes after numbers and bounds.

By the end of this chapter

You will learn the two models — a static machine (a list of live borrows) and a dynamic machine (borrow stacks) — and how a program is reduced to a sequence of three kinds of events. You will understand the central theorem "programs passing the static check have no borrowing violations at run time", its contrapositive, and the two invariants its proof needed. You will also see the story of widening the model to places and op boundaries, the shapes rejected and accepted, and the exhaustive checking of about ten million cases backing the gap between model and implementation. Loops are covered in chapter 43.

The questions this chapter answers

1 If the place model is proven, can `mut_ref (field s a)` and `mut_ref (field s b)` be used at the same time?

42.1 Two machines and three events

Two machines look at the same rule. The **static** machine holds a list of live borrows at compile time and rejects attempts to create overlapping borrows. The **dynamic** machine keeps a borrow **stack** per place at run time and stops when an invalidated borrow is used. This chapter's theorem is that the two machines do not disagree.

For the proof, a program is reduced to a sequence of three kinds of events. Making a small object keeping only the essence instead of the whole language is the first skill of proof.

Event	Meaning
`Create t x mut`	Create a borrow of place `x` with tag `t`. `mut` says whether it can write
`Use t wr`	Access through borrow `t`. `wr` says whether it is a write
`Own x wr`	The owner accesses place `x` directly

Table 42.1 — Three events

The dynamic machine is defined by five rules. Creating a borrow pushes it onto the stack (D1); **reading** through a borrow discards only the write borrows above it and keeps read borrows (D2); **writing** through a borrow discards everything above (D3). An owner read invalidates all write borrows (D4), and an owner write empties the stack (D5). The difference between D2 and D3 is the point — many readers can coexist, but after a write the names above see stale values and must no longer be usable. And the owner touching the place again is not itself an error; the error is caught when a dead borrow is **about to be used** afterwards.

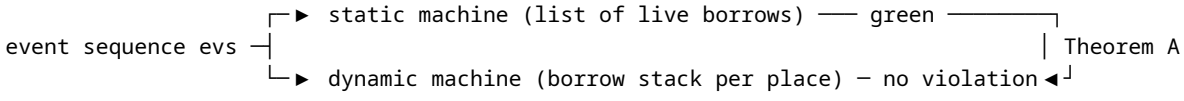
Feeding four events in order changes the stack of place `x` like this.

```
event                       stack (bottom → top)
Create r1 x read    D1      r1
Create w2 x write   D1      r1 w2
Use r1 read         D2      r1          drops the write borrow w2 above r1
Use w2 write                → violation w2 is no longer on the stack
```

The static machine already refuses at the second event: it creates a write borrow on a place where the read borrow `r1` is alive.

42.2 The central theorem

> **The mathematics. Theorem A — agreement (`agreement` in `LowentEXCL.v`)**
>
> `forall evs, wf evs -> static_green evs = true -> dyn_clean evs = true.` For every well-formed event sequence, if the static check passes, the dynamic machine has no violation. Read by contraposition (`no_violation_in_green`): if a borrowing violation happened at run time, the static check certainly rejected that program.




```
static green ⇒ no dynamic violation      (proven)
no dynamic violation ⇒ static green      (not true — the static check is conservative)
```

The contrapositive sharpens the practical meaning. **Code that passed translation never shows a borrowing violation at run time.** The combination "it translated, but running it broke a borrow" is impossible within this model.

The converse — dynamically safe implies statically passing — is not true and need not be. The static check is conservative and may reject programs that are actually safe. Being wrong on the safe side is an inconvenience; being wrong on the dangerous side is a disaster. It is the same asymmetry as chapter 41's interval analysis.

The proof is invariant preservation, and it needed two invariants.

- **INV** — borrows alive statically are also on the dynamic stack, with matching target place and write flag.
- **SINV** — any two distinct borrows alive on the same place are both read borrows.

Why was SINV needed? Writing through a borrow (D3) discards everything above on the stack, and if something discarded is still alive statically, INV breaks. The rule "only one write borrow" prevents overlap, so there is no other borrow alive on the same place as the writing borrow. Intuition says "because overlap is dangerous", but the proof gives a more exact reason — **without that rule the correspondence between the two machines collapses.** Loosen the rule and the proof points to exactly where it breaks.

The rest are tedious lemmas about data structures — "what you store in a place is what you get back" (`get_set_same`), "other places do not change" (`get_set_other`), "discarding above the stack keeps that tag" (`in_pop_above`). Proofs grow long not in the interesting places but in places like these.

42.3 Shapes rejected and shapes accepted

Translated into code shapes, what the theorem stops looks like this. All four were seen with real diagnostics in chapter 12's examples.

Shape	Verdict	In the event model
The owner writes while a write borrow lives	Rejected (`E-EXCL`)	D5 empties the stack, so later use of the borrow is a violation
The same value is borrowed for writing twice	Rejected (`E-EXCL`)	Overlapping borrows including a write break SINV
Writing through a read borrow	Rejected (`E-TYPE-REF`)	A read borrow cannot produce a write event
Several read borrows read together	Accepted	D2 does not kill read borrows — there is no danger

Table 42.2 — Borrow shapes and the theorem

42.4 Finding an implementation defect while widening the model

The flat model saw places as numbers. The real language is more complex. The model was widened twice, and both times Theorem A was proven again.

Places. In the flat model `x.a` and `x.b` are both just `x`, so two write borrows of different fields come out as conflicting. The real rule passes them. A proof that cannot follow the implementation is about a different language. So places were generalised to paths, and **equality** of places was replaced by **overlap** ("one is a prefix of the other"). The structure of the theorem and invariants stayed the same. With a good abstraction, extension ends this locally.

Op boundaries. The event language had no calls. So the shape where an op returns a borrow received as an argument directly as a reference, **laundering** the borrow, was outside the model. The implementation really hit it — the static check did not recognise the returned reference as the same as the original borrow and passed it. After the defect was fixed, the story was turned into a theorem.

> **The mathematics. The old rule was unsound (`LowentLaunder.v`)**
>
> `old_rule_is_unsound` — in the laundering case the old static check passes while the dynamic machine reports a violation. `new_rule_catches_it` — the fixed rule rejects that case. And `honest_laundering` — honest returns still pass. Without the last theorem, "reject everything" would also be a correct answer.

Until then it was a **story**: "we ran into it". Stories are forgotten and erased by refactoring. Now it is a **theorem**. If anyone removes that check, the proof breaks. This is the most valuable thing proofs do in this repository.

> **Q. If the place model is proven, can `mut_ref (field s a)` and `mut_ref (field s b)` be used at the same time?**
>
> **A.** The static check passes it. But this edition's intermediate representation cannot yet lower references pointing at non-local places (fields), so running it says it cannot with `E-IR-UNSUP` and `E-VM-UNSUP`. A proven rule and an implemented feature are different things. Moreover, this edition's `--check` prints that diagnostic and still ends with "check: ok"; being green while an error code is present is a defect.

42.5 Between model and implementation

The theorems are about two machines on paper. Whether those machines match the real compiler is backed by **bounded exhaustive model checking**. All 10,490,024 event sequences with at most two places, four tags and length 7 are fed to the implementation's static check and dynamic machine to see whether the verdicts match the model. Violations: 0.

Length	Event sequences	Time	Violations
6	920,918	4.3 s	0
7 — default	10,490,024	5.0 s	0
8 — opt-in	119,892,870	13.4 s	0
9 — opt-in	1,366,571,808	99 s	0

Table 42.3 — The envelope of exhaustive checking (measured by total unit-test time)

Widening the envelope elevenfold cost under a second. Before writing "too expensive to do", measure. Length 10 and beyond is still **unknown** — and writing that down is this section's job.

Branch merging in loops is outside the theorems, and the loop checker exhaustively confirms 90,376 shapes × 1 … 3 iterations (chapter 43). These are "exhaustively checked", not "proven".

> **A common misconception. Zero violations in exhaustive checking means it is proved**
>
> Exhaustive checking speaks only **inside its envelope**. Even with 1.3 billion sequences up to length 9 and zero violations, length 10 is unknown. That is why this chapter labels theorems A and B "proved" and the correspondence between model and implementation "exhaustively checked", separately. Calling both by one name lets defects outside the envelope hide in "proved territory". The defect found while writing this manual, where lending a `let` name with `mut_ref` changes its value (chapter 12), lies where neither the theorem nor the exhaustive check reaches, because the event model has no notion of an immutable name at all.

42.6 What is not proven

- **The model is a straight sequence of events.** There are no branches or loops. The real compiler computes the life of borrows along control flow, and Theorem A does not say that computation is right. That is the subject of chapter 43.
- **Tags are assumed fresh**. The implementation does not reuse tags, but that fact itself is outside the proof.

- **Memory itself was not modelled**. The event model only looks at "who has the right to access". The step from that system of rights to actual memory safety such as use-after-free is separate.
- **It is a theorem about the sequential fragment**. The extension to several flows is handled by the data race theorem in chapter 45.

Recap

Comparing a static machine (a borrow list) and a dynamic machine (borrow stacks) over sequences of three events proved Theorem A: programs passing the static check have no borrowing violations at run time. Read by contraposition, code passing translation has no borrowing violations at run time. The proof needed the correspondence of the two machines (INV) and "overlapping borrows are all reads" (SINV). Widening the model to places and op boundaries turned the implementation's laundering defect into a theorem. About ten million cases of exhaustive checking back the gap between model and implementation.

43 Loops and fixed points — proving "however many times"

In this chapter

43.1 It breaks on the second round 498
43.2 At a fixed point, any number of times 499
43.3 Only one dangerous shape 500
43.4 Fresh tokens — one move that removes the naming problem 500
43.5 Nested loops — from function to relation 501
43.6 What is not proven 502

What to know first

chapter 7, Flow · `while` repeats its body while the condition holds
chapter 39, The mathematical toolkit · a fixed point is a point that applying once more does not change
chapter 42, Proofs about ownership and borrowing · Theorem A is about a straight sequence of events

Looking back

What was the first line of "What is not proven" for Theorem A in chapter 42?

A. That the model is a straight sequence of events, with no branches or loops. The real compiler looks at code **once**, and that code may run **a million times**. One check must guarantee a million runs. This chapter is how, and the answer is one word — fixed point.

The need for this chapter, and its context

Why loops are dangerous is clear. Defects that are fine when run briefly but break when run long are the hardest to find, because tests run briefly. And inside a loop the same name points to a different thing each round — the borrow in round 1 and the borrow in round 2 are the same `r` in the source but different borrows at run time. This chapter solves both difficulties with fixed points and fresh tokens, and shows there is **exactly one** shape in which borrows break in loops. It is a case of knowing how many rules are needed by argument rather than by feel.

By the end of this chapter

You will confirm with examples the shape where a borrow living across a loop breaks on the second round, and the safe shape where a borrow is born and ends within a round. You will learn Theorem B — if the static state returns after one round, it is safe however many times it runs — the method of issuing a new token each round to remove the naming problem, and the invariant FRESH added to the proof. You will also see the generalisation to nested loops that run differently each round by writing unfolding as a relation, and why termination is not covered.

The questions this chapter answers

1 How does the compiler find the fixed point? Does the theorem guarantee that too?

43.1 It breaks on the second round

examples/ch43/loop_stale.low

```
module loop_stale .
rem expect: E-EXCL

proc drift input n u64 . output u64 .
  requires le n 100 .
do
  var x u64 be 1 .
  let r ref u64 be ref x .
  var i u64 be 0 .
  var seen u64 be 0 .
  while lt i n . do
    set x (add x 1) .
    set seen (deref r) .
    set i (add i 1) .
  end
  return seen .
end
```

Output

```
$ lowentc --check loop_stale.low
12:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
12:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
```

A read borrow `r` created outside the loop is used inside it, while inside the same loop the owner writes to `x`. Round 1 looks fine. But that write kills `r`, and round 2's `deref r` uses a dead borrow. It is fine when run once and blows up the second time. The processor rejects it at translation.

examples/ch43/loop_fresh.low

```
module loop_fresh .
rem run: drift 5

proc bump input p mut_ref u64 . output u64 .
do
  set p (add (deref p) 1) .
  return deref p .
end

proc drift input n u64 . output u64 .
  requires le n 100 .
do
  var x u64 be 1 .
  var i u64 be 0 .
  var seen u64 be 0 .
  while lt i n . do
    set seen (bump (mut_ref x)) .
    set i (add i 1) .
  end
  return seen .
end
```

Output

```
$ lowentc --run drift loop_fresh.low 5
drift(5) = 6
```

Conversely, if the borrow **is born and ends inside** the loop, it is fresh every round, and nothing stale remains between rounds. The common advice "create it inside the loop and finish it inside the loop" has a proof attached.

43.2 At a fixed point, any number of times

Here the function `f` is "running the loop body once", and the value `x` is **the static checker's state** (the list of live borrows). If one run of the body leaves the static state unchanged, that is a fixed point. Then induction holds.

```
Running 0 times is safe.                                  base --- nothing was done
Running once safely from state L returns the state to L.  preservation
-------------------------------------------------------
Therefore it is safe however many times it runs.          ∀k
```

It is like dominoes needing equal spacing. If the state changed every round, there would be no argument for “the next piece”. Because the state returns, the same argument can be reused endlessly.

> **The mathematics. Theorem B — loop agreement (`loop_agreement` in `LowentLoop.v`)**
>
> `forall pre body, (exists lpre, srun [] pre = Some lpre /\ srun lpre body = Some lpre) -> forall k, dyn_clean pre body k = true.` If, from the static state after setup code `pre`, running the loop body `body` once **returns to the same state**, the loop has no borrowing violation however many times it runs. `forall k` is the whole theorem — 0 times, once, 2^{64} times. `k = 0` is included. It looks trivial, but leave it out and the induction fails.

Some code does not reach a fixed point. If a borrow is added each round, never dies, and escapes the loop, the static state grows. Such code does not satisfy the theorem’s premise, and the processor rejects it. When a borrow is born and dies in the body, the state returns — that is ordinary code.

43.3 Only one dangerous shape

Something was learned while proving. There is **only one way** borrows break in loops.

```
A borrow τ created in pre is used in body, while body also contains owner access own(x).
  → own(x) in round i kills τ.
  → use(τ) in round i+1 is a violation.
```

One static rule rejects exactly that pair — a borrow living across the loop × owner access in the body — and the proof of Theorem B shows “that one is enough”. There is no other danger. This is the proof’s **practical output**. Too few rules leak defects and too many block normal code, and the argument decided how many is right.

43.4 Fresh tokens — one move that removes the naming problem

The real headache in loops is **name clashes**. The `r` of round 1 and the `r` of round 2 are the same name in the source but different borrows at run time. Handling that usually means building a “tag renaming” device.

This proof built no such device. Instead the model **issues a new token every time it runs**. A static tag `t` points through an environment to “the current token”, and that token gets a new number each round. Then last round’s token is pointed to by no one and becomes invalid by itself. This is no trick — **the implementation really issues fresh tags.** Because the model followed the implementation, the proof got shorter. If a proof is hard, look at the model again. Half the difficulty comes from a wrong representation.

The proof adds one invariant to chapter 42′s INV and SINV.

- **FRESH** — every issued token is smaller than `next`.

Without it a new token could receive the same number as an old one, and a dead borrow would **come back to life**. It is the scariest kind of hole in a proof — wrong on the dangerous side, not the safe side.

43.5 Nested loops — from function to relation

examples/ch43/nested.low

```
module nested .
rem run: tri 4

proc bump input p mut_ref u64 . output u64 .
do
  set p (add (deref p) 1) .
  return deref p .
end

rem the inner loop runs a different number of times each outer round (0, 1, 2, 3)
proc tri input n u64 . output u64 .
  requires le n 100 .
do
  var x u64 be 0 .
  var i u64 be 0 .
  while lt i n . do
    var k u64 be 0 .
    while lt k i . do
      let seen u64 be bump (mut_ref x) .
      set k (add k 1) .
    end
    set i (add i 1) .
  end
  return x .
end
```

Output

```
$ lowentc --run tri nested.low 4
tri(4) = 6
```

The inner loop runs 0, 1, 2 and 3 times on successive outer rounds. The borrow `mut_ref x` is born and ends in the inner body, so it is safe in every round.

The flat Theorem B imposes `unroll body k` — "every round runs the same". Real programs do not. The nested loop proof (`LowentNest.v`) represents the program as a tree (`NLeaf`, `NSeq`, `NLoop`) and writes unfolding **as a relation, not a function**.

> **The mathematics. Nested loops (`nested_agreement`)**
>
> A nested loop program passing the static check has no borrowing violation **under any unfolding**. There is no depth limit, no iteration limit, and **rounds may run differently**. Written as a relation, "the inner loop runs three times in round 1 and zero times in round 2" is simply another unfolding of the same tree. And the old flat theorem is recovered as a corollary (`flat_loop_recovered`). If it could not be recovered, it would not be a generalisation but a different theorem.

Loops whose first round changes the state fit too. Peeling once as `b ; loop b` leaves a fixed point afterwards (`peeled_loop_is_safe`).

Q. How does the compiler find the fixed point? Does the theorem guarantee that too?

A. The theorem **assumes** the fixed point at the entry. The compiler **reaches** it by merging the states of branches and rounds (with the lattice's join and monotone iteration), but that process is outside the model. The gap is backed by bounded exhaustive checking — the loop checker runs all 90,376 shapes for rounds 1 … 3 and checks that model and implementation agree. This is "exhaustively checked", not "proven".

A common misconception. If following the loop body once shows no problem, any number of rounds is fine

`loop_stale.low` is the counterexample: round 1 is fine, and round 2 uses a dead borrow. "Following it once" is enough only when the static state after one pass **equals the state before**, that is, at a fixed point. So the checker applies the body repeatedly until the state stops changing, not just once, and this chapter's theorem guarantees that stopping there is correct. The same holds when a person reads code: ask "what is alive when the second round begins?"

43.6 What is not proven

- **Merging `if` branches is not in Theorem B.** The model is two straight sequences `(pre, body)`. The correctness of merges is backed by the exhaustive checking above.
- **The process of reaching the fixed point** is outside the model. The theorem speaks of after it is reached.
- **Whether loops terminate is not covered.** The theorem is "safe if it runs", not "it ends". Infinite loops, like a firmware main loop, are legitimate programs in this language. That is why tests the tool generates for itself carry a step budget (chapter 41) — termination cannot be guaranteed, so the tool must know when to stop.

Recap

A borrow living across a loop meeting owner access in the body breaks on the second round, and the processor rejects that one pair. A borrow born and ending in the body is fresh every round. Theorem B — if the static state returns after one round (a fixed point), there is no borrowing violation however many times it runs — is induction on `k`, and a model issuing new tokens each round with the invariant FRESH removes the naming problem. Nested loops were covered, including rounds that run differently, by writing unfolding as a relation. Merges and reaching the fixed point are backed by exhaustive checking, and termination is not covered.

44 Proofs about effects — writing "what it can do" into the type

In this chapter

44.1 Effects are sets of atoms 504
44.2 Four rules the processor enforces 504
44.3 Capabilities — the partner of effects 506

44.4 Effect soundness — not circular 506
44.5 Optimisations relying on effects are legitimate 507
44.6 What is not proven 508

What to know first

chapter 15, Effects · effects spread along calls, and purity permits reordering, remembering and deleting

chapter 16, Capabilities · effects write what kind of work, capabilities write who permitted it

chapter 39, The mathematical toolkit · lattices and join

Looking back

In chapter 15, what separated `fn` from `proc`, and what did the promise of purity permit the processor to do?

> **A.** `fn` leaves no trace outside and `proc` leaves as much as its written effects. A pure op gives the same answer for the same arguments, so the processor may reorder calls, remember results, or delete unused calls. This chapter covers the proof that the promise **really holds** — and why that proof is not circular.

The need for this chapter, and its context

chapters 40 and 43 were about “how memory is touched”. This chapter is about “what it can do”. `effects none` is a strong promise — it reads no globals, allocates nothing, writes nothing to the screen, waits for nothing. If the optimiser trusts that promise and deletes a call that actually wrote a file, observable behaviour disappears. The effect system does not give memory safety. Instead it gives a different kind of safety — legitimate optimisation, auditability, rejection on small machines. That safety comes from trusting declarations, so a theorem that declarations cover reality is needed.

By the end of this chapter

You will learn that effects are sets of atoms forming a lattice under subset order, and that propagation and gating are two operations of one lattice. You will confirm with examples the four rules the processor enforces (propagation · purity · a set of atoms · stopping is not an effect), and pick up attenuation and local audit, which capabilities give as the partner of effects. You will also see why the effect soundness theorem is not circular, and the proof, with counterexamples, that four optimisations relying on effects are legitimate.

The questions this chapter answers

1 Why does using a reserved word as a parameter name give a name diagnostic rather than an effect diagnostic?

44.1 Effects are sets of atoms

An effect is **a set of atoms**. It can hold several atoms, as in `effects io alloc` .. Then there is a natural order — $\varepsilon_1 \sqsubseteq \varepsilon_2$ is $\varepsilon_1 \subseteq \varepsilon_2$, "$\varepsilon_1$ does less than ε_2". The subset relation forms a lattice, the join is union, and `none` (the empty set) is the bottom.

What	Lattice operation	Meaning
Propagation	$\sqcup$ (union)	Gathering the effects of everything I call gives my effect
Gating	$\subseteq$ (subset)	Doing less than declared is fine

Table 44.1 — Two jobs done by one lattice

It is **the same shape** as chapter 40's type lattice. The same mathematics answers a different problem. And the judgement on small machines becomes one $\subseteq$. Code that allocates must not translate for a machine without a heap (chapter 30), and with effect declarations it is enough to ask "is this op's effect within the set this tier supports".

```
            {io, alloc}
             ▲      ▲
            {io}    {alloc}
             ▲      ▲
              none

propagation  if f calls g (effects io) and h (effects alloc),
             effects(f) ⊇ {io} ⊔ {alloc} = {io, alloc}      short of it: E-EFFECT
gating       if the set a heapless board supports has no alloc,
             {io, alloc} ⊄ that set                        it does not translate for that board
```

A **hierarchical** lattice splitting `io` into `{read, write}` is possible too. This language uses a **flat** one for now. The precision a hierarchy gives is already available from capabilities, and a flat lattice keeps the $\subseteq$ test simple so the cost is visible. Express precision in two places and the two begin to diverge.

44.2 Four rules the processor enforces

Rule 1 — propagation. If `f` calls `g`, then `effects(f) ⊇ effects(g)`. Otherwise it is `E-EFFECT` (`E-EFFECT-CALC` for a `fn`). It stops an op marked `effects none` from secretly writing a file (chapter 15).

Rule 2 — purity. A `fn` writing to the caller's buffer is `E-EFFECT-PURITY`. `fn` computes values and `proc` does work. If the criterion is **the shape of a word**, holes appear. At one time an actor's handlers were counted as pure because they were not `proc`, so a handler mutating actor state could declare `effects none` and nobody objected. The same offence was caught on one side and not the other.

Rule 3 — an `effects` clause is a set of atoms.

examples/ch44/effect_dup.low

```
module effect_dup .
rem expect: E-EFFECT-DUP

proc shout input out cap io . output u64 . effects io io . do
```

```
  return write_out out 1 "hi\n" .
end
```

Output

```
$ lowentc --check effect_dup.low
effect_dup.low:4:0 E-EFFECT-DUP: an effect atom appears more than once in the `effects`
clause — the row is a SET, not a list. A repeat says nothing the first one didn't, and t
he reader has to decide it's noise (RFC-0057 E3)
```

examples/ch44/effect_typo.low

```
module effect_typo .
rem expect: E-EFFECT-UNDEF

proc ping input out cap io . output u64 . effects netwrok . do
  return 0 .
end
```

Output

```
$ lowentc --check effect_typo.low
effect_typo.low:4:0 E-EFFECT-UNDEF: unknown effect (the vocabulary is closed: none/alloc
/heap/io/wait/concurrent/lock/atomic/unsafe/device/page_fault/blocking/cancel/detach/pan
ic/state) — a typo here silently declares the op PURE
```

Writing the same atom twice is rejected. A duplicate in a set means nothing, and the reader has to decide whether it is noise. A nonexistent atom is caught as a typo — as the diagnostic says, a typo passing silently would **declare the op pure**. The rule looks obvious, but there was a time the implementation read only **the first word** of the clause. Then `effects io alloc` registered only `io`, and atoms from the second on were as good as absent. A defect in the code reading declarations disables the whole declaration system.

Rule 4 — stopping is not an effect. This is the most thought-provoking part of the chapter.

examples/ch44/trap_not_effect.low

```
module trap_not_effect .
rem run: pick [1,2,3] 1
rem trap: pick [1,2,3] 7

rem it stops when out of range, but this op is still a pure fn
fn pick input s slice u8 . input i u64 . output u8 . do
  return index s i .
end
```

Output

```
$ lowentc --run pick trap_not_effect.low [1,2,3] 1
pick([1,2,3], 1) = 2
  arg0 (written) = [1,2,3]
$ lowentc --run pick trap_not_effect.low [1,2,3] 7
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

`index s i` stops when out of range. So should `pick` carry the `panic` effect? **No.** `panic` counts only an explicit `panic "…"`. The possibility of stopping is what **contracts** speak of, not effects. The danger of `index` **can be removed** with `requires lt i (len s)`, and the processor really does remove that check (chapter 41). An effect mark is never removed. Marking something removable as an effect makes the mark permanent, nearly every op ends up with `panic`, and the mark loses meaning.

```
what ends the op              where it is written       can it disappear
panic "…"  (intended)         effects panic             no — it always stays
index out of range (breach)   requires lt i (len s)     the check goes once proven
```

> **A common misconception. An op that can stop is not pure**
>
> Effects say “what this op does” and contracts say “when it is safe”. Because the two were not mixed, the `panic` mark still carries information. An op marked `panic` can end the program **on purpose**. An out-of-range index is not intent but a contract violation, and contracts handle that place.

44.3 Capabilities — the partner of effects

If effects say “what kind of work”, capabilities say “where the **right** to do it came from”. Writing to the screen requires receiving `cap io` as an argument; it cannot be pulled out globally from anywhere. This object-capability model gives two things.

- **Attenuation.** A received right can be narrowed before passing it on. The precision a flat effect lattice lacks is obtained here.
- **Local audit.** Looking at an op’s signature shows **everything** it can touch. That there are no hidden global passages is the point.

What it gives	How
Legitimate optimisation	With `effects none`, reordering, common subexpression elimination and deletion do not change observable behaviour
Auditability	Code doing `io` can be found from declarations
Tier and profile gating	Ops allocating on small machines are stopped as translation errors
No surprises	What is written as pure really is pure

Table 44.2 — Safety the effect system gives

44.4 Effect soundness — not circular

examples/ch44/recursive.low

```
module recursive .
rem run: main

proc stars input out cap io . input n u64 . output u64 . effects io .
  requires le n 10 .
do
  if eq n 0 . do
    return write_out out 1 "\n" .
  end
```

```
  let w u64 be write_out out 1 "*" .
  return add w (stars out (sub n 1)) .
end

proc main input out cap io . output u8 . effects io .
do
  return narrow u8 (stars out 5) .
end
```

Output

```
$ lowentc --run main recursive.low
*****
main() = 6
```

`stars` is recursive, calling itself, and declares `effects io`. At a call site the processor uses the callee's **declaration** rather than walking its **body** again. At first glance this looks circular — a declaration trusts a declaration.

> **The mathematics. Effect soundness (`effect_sound` in `LowentEffect.v`)**
>
> `prog_ok P = true -> lookup P f = Some (d, b) -> performs P (ECall f) a -> In a d.` In a program passing the check, when op `f` is called, every effect that actually happens is within what `f` declared. Call depth does not matter. It is not circular because the induction is over **executions**, not **programs**. When an effect happens, it happens at some finite call depth, and induction on that depth closes. So recursion, mutual recursion and non-terminating recursion need no separate treatment. The proof file includes an example where a self-calling op passes the check, `io` actually happens, and `alloc` **cannot** happen.

44.5 Optimisations relying on effects are legitimate

The legitimacy of optimisations stands on effect soundness. At the effect layer, pure pieces may be reordered (`pure_calls_commute`) or deleted (`dropping_pure_is_legal`) with the same effects happening. The proof raised to the value layer is `LowentOpt.v`. In a model with values, state and an output stream, reordering, common subexpression elimination, memoisation and dead code elimination were all four proven legitimate.

The key is one line — **a value depends only on the places it reads** (`value_depends_only_on_reads`). And counterexamples showed the premises are not decoration. Reordering effectful pieces **changes** a value from 14 to 7, and deleting a piece that prints shrinks the output. One theorem, one counterexample — without the counterexample, the theorem would be true even if the condition were overly strict.

> **Q. Why does using a reserved word as a parameter name give a name diagnostic rather than an effect diagnostic?**
>
> **A.** At one time it was the other way round. Naming a parameter `raw`, the raw-pointer word, silently contaminated effect inference and produced "declared pure but performs an effect" (`E-EFFECT-CALC`). The real reason was "a reserved word was used as a name", but the diagnostic

pointed at purity. A diagnostic that sends the programmer to the wrong place is the worst kind. So those words were handed to `E-NAME-BUILTIN`, leaving one diagnostic for one offence. Proofs decide what is rejected, but people meet why — if the latter is wrong, the former loses half its value.

44.6 What is not proven

- **That the implementation matches the model** is an empirical correspondence backed by tests and back-end cross-checks (chapter 50).
- The rule by which `task_group` **absorbs** `concurrent` is not a $\subseteq$ discipline but discharging a requirement, so it is outside the model.
- In effect-based optimisation, **call boundaries** (raising the purity judgement to op boundaries) and **the moment of stopping** are outside.
- **The precision limit of a flat lattice.** Refinements like `io ⊑ {read, write}` were not done, and capability attenuation stands in for them, with no argument that the two give the same precision.
- **No effect-row polymorphism.** An op "inheriting the caller's effects as they are" cannot be written, so higher-order ops must declare the widest effects.
- **Some atoms are vocabulary only.** `page_fault`, `blocking`, `cancel`, `detach` and others can be declared but have no operation producing them yet, so propagation of those atoms is thinly tested in practice.

Recap

Effects are sets of atoms forming a lattice under subset order; propagation is union and gating is a subset test. The processor enforces four rules — propagation · purity · a set of atoms (rejecting duplicates and typos) · stopping is not an effect — and capabilities, as the partner of effects, give attenuation and local audit. Effect soundness is induction over executions, so it is not circular even with recursion, and on top of it reordering, common subexpression elimination, memoisation and dead code elimination were proven legitimate along with counterexamples.

45 Proofs about races and parallelism — discipline instead of a memory model

In this chapter

45.1 The Bernstein condition — an answer from 1966 509
45.2 Independence means no races, and the condition is necessary 510
45.3 Between actors there is always a message 511
45.4 Containment — where heavy tools are needed 511
45.5 No races is not the same as deterministic 511
45.6 What is not proven 513

What to know first

chapter 25, Actors · state is locked inside the actor and ownership moves by message
chapter 27, Parallel loops and atomic operations · the three conditions for a splittable loop, and `reduce`
chapter 42, Proofs about ownership and borrowing · within one flow, only one name writes

Looking back

In chapter 27, why did the `W-PAR-OK` note say that the VM running sequentially is **a correct implementation**?

> **A.** Because a theorem is proven that when the splittable conditions hold, the parallel result is bit-for-bit identical to the sequential one. This chapter covers that theorem and the one that must stand before it — "there are no data races". They are different conclusions from the same premise.

The need for this chapter, and its context

When two flows touch the same place at the same time and one of them writes, it is a data race. In C and C++ it is undefined behaviour. The usual way to deal with races is to learn a memory model such as memory orderings, which is hard. This language claims something else — **code in split loops (level 1) and actors (level 2) never needs to look at a memory model.** Races are ruled out by discipline. And a guarantee that answers are the same every time, even without races, is needed separately. This chapter is the story of establishing both claims in pure Coq without heavy concurrency logic. Reading the proposition exactly reduces the tools needed.

By the end of this chapter

You will learn the three-line set condition called Bernstein independence and the definition of a race. You will pick up the pair of theorems that at level 1 independence means no races and the condition is necessary, and the theorem that at level 2 a message always lies between accesses by different actors. You will also see why having no races differs from being deterministic, DET-1 (parallel results are bit-for-bit identical to sequential), DET-2 (freedom for the scheduler), DET-3 (associativity frees the shape of the reduction tree), and the counterexample for each.

The questions this chapter answers

1 Isn't "roughly the same answer" good enough?

45.1 The Bernstein condition — an answer from 1966

Let `R` and `W` be the **set of places read** and **set of places written** by a task. Two tasks are **independent** in three lines.

```
W₁ ∩ W₂ = ∅      the two do not write the same place
W₁ ∩ R₂ = ∅      one does not read what the other writes
R₁ ∩ W₂ = ∅      nor the reverse
```

Three lines saying intersections are empty are all there is. `R₁ ∩ R₂` is not in the condition — **reading** the same place together is fine. A race is defined as "two tasks touch the same place and at least one writes". The usual definition adds "the two accesses are not ordered", but at level 1 that holds automatically. Between tasks after flows split and before they rejoin, there are no ordering edges at all.

The processor's diagnostic quotes this condition directly.

examples/ch45/par_overlap.low

```
module par_overlap .
rem expect: E-PAR-WRITE

rem every piece writes the first cell --- what remains depends on who went last
proc stamp input s mut slice u64 . output u64 . effects none .
  parallel s split .
do
  var i u64 be 0 .
  while lt i (len s) . do
    set (index s 0) i .
    set i (add i 1) .
  end
  return len s .
end
```

Output

```
$ lowentc --check par_overlap.low
par_overlap.low:10:0 E-PAR-WRITE: a splittable loop may only write its OWN element `inde
x <s> <i>` — this write can collide with another iteration (Bernstein: wr ∩ wr = ∅)
par_overlap.low:10:0 E-PAR-READ: a splittable loop may only read its OWN element of the
split slice — reading another index creates a cross-iteration dependence (Bernstein: rd
∩ wr = ∅)
```

Every piece writes the first element, so `wr ∩ wr = ∅` broke, and reading another piece's element broke `rd ∩ wr = ∅` too. chapter 27's three conditions — read only your own share, write only your own share, do not write places living across iterations — are the Bernstein condition.

45.2 Independence means no races, and the condition is necessary

> **The mathematics. Level 1 — split loops (`LowentDRF.v`)**
>
> `l1_no_race : forall t1 t2 l, indep t1 t2 -> ~ races t1 t2 l.` If two tasks are Bernstein independent, they race at **no place**. The proof is five lines. Assuming a race means "one writes", and the independence condition denies the other's access to that place in all three cases. Contradiction.
>
> `l1_condition_is_necessary : ~ indep writer reader /\ races writer reader 0.` Actually build a pair breaking the condition — one writes place 0 and the other reads place 0 — and a race **exists**.

The second theorem is the mark of an honest proof. Proving only “satisfying the condition is safe” leaves the theorem true even if the condition is overly strict — a condition that lets nothing pass is also safe. So **a counterexample is proven alongside**: loosen the condition and it really breaks. One theorem, one counterexample — a method this part uses in many places.

Without races, the well-known DRF-SC theorem (Adve–Hill) applies and the program behaves with sequential consistency. That is, you need not imagine orders jumbled up.

45.3 Between actors there is always a message

> **The mathematics. Level 2 — actors (`12_accesses_are_separated_by_a_message`)**
>
> If in an execution trace **different** actors access the same place `l`, then **between** the two accesses there is always a message handing over ownership of that place.

The chain of argument runs like this. A place is owned by exactly one actor. Only the owner accesses it. Ownership moves only by message. But the two accesses come from different actors. Therefore an ownership transfer happened between them. A message is an edge fixing before and after, and two accesses with a fixed order are **by definition not a race**. The most common mistake with actors — sending a reference by message and continuing to use it on the sending side — is stopped because it is an ownership transfer and the sender loses that place (chapter 25’s `handoff`).

The connection to chapter 42 is the point. Enforcing “only one name writes” within one flow becomes exactly “two tasks do not overlap”. One safety rule pays off in two places.

45.4 Containment — where heavy tools are needed

```
level 1 (split loops)          ─┐
level 2 (actor isolation)       ─┼→ no races by discipline → DRF-SC → think sequentially
level 3 (atomics · locks)       ─┘→ races really exist     → a memory model is needed
```

> **A common misconception. Proving concurrency always needs heavy tools like separation logic**
>
> The first plan was to leave “safe code has no races” as a sketch and do the machine proof with heavy concurrency logic. Looking again, that proposition was not a claim about memory models. “Discipline prevents races” is a **set-theoretic** claim. Heavy tools are really needed only at level 3, and among the proof files only the lock uses Iris (chapter 47). “It is concurrency, so separation logic” was a reflex.

45.5 No races is not the same as deterministic

Running twice on the same input can give different answers. Even without races. If several cores add floating-point numbers in parts and combine them, the last bits change when the **order** of combining changes. Each touched only its own share, so there is no race. Yet the answers differ.

This language's claim is strong. **The result of level 1 parallelism is bit-for-bit identical to the sequential result.** No approximation, no tolerance. The conditions establishing that claim were proven to be exactly three (`LowentPar.v`). Determinism is not a matter of races but of the Bernstein condition. The same condition gives a different conclusion.

Condition	With it	Without it
DET-1 split without overlap	Parallel = sequential (`det1_par_eq_seq`)	Order changes the result (`overlap_is_nondeterministic`)
DET-2 keep dependency edges	The remaining order is free (`det2_schedule_free`)	— not using the freedom only loses performance
DET-3 associative combining	Tree shape does not change the result (`assoc_shape_free`)	Same leaves, different results (`nonassoc_shape_matters`)

Table 45.1 — Three conditions for deterministic parallelism — with and without

> **The mathematics. DET-1 — parallel equals sequential**
>
> If tasks are pairwise independent and each is **local** (decides values only from its own read set), the value at **every place** `i` after parallel execution equals sequential execution. The corollary's name is the claim itself — `det1_bit_identical`. The fork in the proof is instructive. If the first task writes place `i`, no one else writes `i`, so the answer is the first task's. If the first task does not write `i`, the one writer among the rest gives the answer, and since that task's **reads** do not overlap the first task's writes, reading the original state or the state the first task touched gives **the same value**. The second case says why "local" is needed.

The counterexample is two tasks writing 1 and 2 to the same place. Depending on order, the answer is 2 or 1.

DET-2 is the theorem that swapping two non-conflicting tasks gives the same result. The proof is **induction on adjacent swaps**. If adjacent pairs can be swapped, repeating that reaches "any rearrangement that keeps the conflict order" — the same idea as a sorting algorithm. It gives the scheduler maximal freedom without losing determinism, and freedom is performance.

DET-3's counterexample is subtraction. `(5 - 3) - 1 = 1` while `5 - (3 - 1) = 3`. If the operation is not associative the language must fix the tree shape, and if the scheduler changes the tree with the number of cores, the result depends on the schedule. The proof did not import IEEE floating point as axioms; it showed "if non-associative, the shape matters" generally, using subtraction as the representative non-associative operation. That is why `reduce acc sub` is rejected with `E-PAR-ASSOC` and floating-point addition with `E-PAR-FLOAT` (chapter 27).

> **Q. Isn't "roughly the same answer" good enough?**
>
> **A.** Letting it pass because the values are almost the same **makes bit comparison unusable for regression tests.** Losing reproducibility loses a debugging tool. Parallel sums giving different answers per core count are the most common cause of irreproducibility in scientific computing. In this language there is no "roughly" — if the three conditions cannot be confirmed, translation refuses.

45.6 What is not proven

- **The DRF-SC theorem itself is cited.** “Without races it behaves with sequential consistency” is a well-known theorem; what was proven here is that our discipline satisfies its **premise** (no races).
- **The theorems do not verify the implementation.** Level 1 parallelism is implemented, but what the theorems pin down are **the conditions the implementation must keep**. Whether it keeps them is backed empirically by back-end cross-checks.
- **The model assumes static read and write sets.** With pointers those sets are hard to know in advance; in practice the type system must guarantee them. “Local” is an assumption, not a check.
- **The state model is a “place → value” function.** Real memory with alignment, aliasing and partial writes was not modelled. Floating point itself (rounding, NaN, −0) was not covered either.

> **Recap**
>
> Bernstein independence is three empty intersections; at level 1 independence means no races and the condition is necessary, both proven with a counterexample. At level 2 an ownership message always lies between accesses by different actors, so races are absent by definition. Heavy tools are needed only at level 3. Having no races differs from being deterministic, and three conditions — splitting without overlap · keeping dependency edges · associative combining — make parallel results bit-for-bit identical to sequential ones. Each condition has a counterexample showing what breaks without it.

46 Proofs about weak memory — with the default, you may think sequentially

In this chapter

46.1 Sequential consistency and its counterexample 514
46.2 An execution is a graph 515
46.3 Impossible with the default, possible with `relaxed` 515
46.4 Exactly that axiom forbids it 516
46.5 Message passing — the weak ones exist for this shape 516
46.6 Over all executions 517
46.7 Weaker orderings add behaviours 517
46.8 Seven tests from the literature 518
46.9 What is not proven 518

What to know first

chapter 27, Parallel loops and atomic operations · the five memory orderings `order` and the default `seq_cst`
chapter 39, The mathematical toolkit · partial orders — orders that allow incomparable pairs
chapter 45, Proofs about races and parallelism · levels 1 and 2 remove races by discipline

Looking back

Where was a memory model needed in chapter 45's containment picture?

> **A.** At level 3 — atomic operations and locks. There two flows really do hammer the same place, so discipline cannot remove races. This chapter looks head on at the weak memory model that decides "what can be seen, and when" there.

The need for this chapter, and its context

CPUs and compilers reorder instructions. With one flow it goes unnoticed, but when several flows look at the same memory it shows — others do not see things in the order I wrote them. Lowent exposes five strengths for atomic operations (`relaxed` < `acquire`, `release` < `acq_rel` < `seq_cst`) and makes the strongest, `seq_cst`, the default. The weak ones must be written to be obtained, and that code is subject to audit. This decision rests on two propositions: that with the default you may think sequentially, and that the weak ones **really** break that. Prove only one and the decision is not justified. Without the first, "default `seq_cst`" is only comfort; without the second, "write it and audit it" is an exaggeration.

By the end of this chapter

You will learn what sequential consistency is and the standard counterexample that breaks it (store buffering), and the RC11 model, which views an execution as a graph with several kinds of edges and judges that an execution with a cycle does not exist. You will pick up the pair of theorems that the counterexample is impossible with the default and possible with `relaxed`, exactly which axiom forbids it, and the theorem that `release` and `acquire` really synchronise message passing. You will also see the general theorem over all executions, monotonicity (weaker orderings add behaviours), and the model defect that proof found.

The questions this chapter answers

1 What happens to an atomic operation with no `order` written?

46.1 Sequential consistency and its counterexample

Sequential consistency (SC) means "all accesses of all flows **can be lined up in one row**, with the order within each flow kept in that row". It is the picture people naturally imagine. Real hardware does **not** behave this way.

```
initially: x = 0, y = 0

flow 1:  x ← 1        flow 2:  y ← 1
         r1 ← y                r2 ← x

bad outcome:  r1 = 0  and  r2 = 0
```

It is the standard counterexample called **store buffering**. Why is it bad? It cannot be lined up. `r1 = 0` puts flow 1's read before flow 2's write, and `r2 = 0` puts flow 2's read before flow 1's write. But within each flow the write comes before the read. Contradiction. Yet **this outcome happens on real x86.**

46.2 An execution is a graph

The reference model is RC11 (Lahav et al., PLDI 2017). It views an execution as a graph. Events are vertices, and there are several kinds of edges.

Edge	Meaning
`hb` (happens-before)	The order is guaranteed
`rf` (reads-from)	This read read that write's value
`mo` (modification order)	The global order of writes to one place
`fr` (from-read)	This read comes **before** that write — it read an older value

Table 46.1 — RC11 edges

The judgement is one sentence. **If a cycle appears on some axis, the execution does not exist.** That is all the mathematics this chapter needs — does a directed graph have a cycle. If you have learned topological sorting, you know the idea. RC11 is hard not because of the idea but because there are many kinds of edges.

46.3 Impossible with the default, possible with `relaxed`

> **The mathematics. E1 — `sb_sc_impossible`**
>
> `consistent (SB SC) rf_00 mo_SB = false.` If every access is `seq_cst`, the outcome "both read 0" violates the consistency axioms — no such execution exists. See the cycle directly: `Wx1 –hb→ Ry0 –fr→ Wy1 –hb→ Rx0 –fr→ Wx1`. To read `fr`: `Ry0` read the initial value of `y`, which `Wy1` overwrote, so `Ry0` comes before `Wy1`. The four edges close a loop, so this execution cannot exist.

> **The mathematics. E2 — `sb_relaxed_is_consistent`**
>
> `consistent (SB Rlx) rf_00 mo_SB = true.` With `relaxed`, that bad outcome **satisfies every axiom** — it can really happen. The sequential consistency axiom speaks only of `seq_cst` events, so with only `relaxed` it is vacuously true. That is both the value and the danger of `relaxed`. Removing constraints makes it fast, but what was removed is exactly the safety.

E1 turns the sentence "default `seq_cst`" from comfort into a **theorem**. Code using only `seq_cst` may be imagined lined up in a row.

46.4 Exactly that axiom forbids it

> **The mathematics. E3 — which axis does the work**
>
> `coherence_alone_does_not_forbid_sb` — even with `seq_cst`, the coherence axis alone does not forbid store buffering. `it_is_the_sc_axiom_that_forbids_it` — what forbids it is the sequential consistency axis.

> **In practice. The machine taught us the model was wrong**
>
> The first model written **let** store buffering pass even with `seq_cst`. Axioms were missing. The prover exposed that, and the axioms were added. The two theorems of E3 pin the lesson down. With which axis does the work written down, the theorem breaks when someone later deletes the axiom. The real value of a proof tool lies here more than in obtaining theorems.

46.5 Message passing — the weak ones exist for this shape

E1 to E3 said “weak is dangerous”. Then why do the weak ones exist? Exactly for this shape.

examples/ch46/handoff.low

```
module handoff_order .

proc publish input k cap atomic . input cells mut slice u64 . output void . effects atom
ic .
  requires ge (len cells) 2 .
do
  atomic_store cells 0 42 order relaxed .
  atomic_store cells 1 1 order release .
end

proc observe input k cap atomic . input cells mut slice u64 . output u64 . effects atomi
c .
  requires ge (len cells) 2 .
do
  let flag u64 be atomic_load cells 1 order acquire .
  guard eq flag 1 . else return 0 .
  return atomic_load cells 0 order relaxed .
end
```

Output

```
$ lowentc --check handoff.low
== check: ok ==
```

Write the data first (`relaxed`), then write the flag with `release`. The other flow reads the flag with `acquire`, and if it is 1, reads the data. The bad outcome is a stale value: “read flag 1, yet read data 0”.

> **The mathematics. E4 — message passing**
>
> `mp_relacq_forbids_stale` — with `release` and `acquire`, seeing the stale value is **impossible**. `mp_relaxed_allows_stale` — with both `relaxed`, it is **possible**. `mp_sync_is_what_forbids_it` — what

> makes the difference is the synchronisation edge `sw`, standing **between** the `release` write and the `acquire` read. The argument: `sw` creates an `hb` from the data write to the data read. But the data read read the initial value and the data write is later by `mo`, so an `fr` in the opposite direction stands. A cycle forms on the same place, so the coherence axis forbids it.

E4 pairs with E1. E1 says "the default is safe", E4 says "writing it synchronises". Both sides of the design decision are on the machine.

> **The mathematics. E5 — `sb_relaxed_has_a_race`**
>
> An execution yielding store buffering's `(0, 0)` **contains a race**. But as chapter 45 showed, levels 1 and 2 cannot produce such executions. So level 1 and 2 code need not read this chapter — weak memory is a level 3 problem only, and containment holds.

46.6 Over all executions

Most of the theorems above end with `vm_compute. reflexivity.` Because `consistent` was defined as a decision function returning true or false, computing that function on a particular execution finishes the proof. It is short, and when the model changes it re-checks automatically. The price is that it is a result about **particular executions**. The standard counterexamples are exactly what the design decision hangs on, so it suffices to justify it — but it did not stop there.

> **The mathematics. If every access is `seq_cst`, it is sequentially consistent (`all_sc_is_sc` in `LowentRC11SC.v`)**
>
> `all_sc E = true -> consistent E rf mo = true -> sc_com_acyclic E rf mo = true.` If every access is `seq_cst` and the execution is RC11-consistent, then `po ∪ rf ∪ mo ∪ fr` has no cycle — for **all executions · all rf · all mo**. In axiomatic memory models this is the standard definition of "this execution is sequentially consistent". The engine of the proof is one line — **a subrelation inherits acyclicity.**

And it went a step further than "no cycle". Instead of arguing abstractly that "acyclic means a total order exists", it **built topological sorting as a function** and proved its result is a genuine linear extension (`topo_linearises`, `sc_witness_orders_everything`). All accesses were laid out in one row, and that row keeps all of program order and communication order. For the good store buffering execution the row `[0; 1; 2; 4; 3; 5]` actually comes out.

46.7 Weaker orderings add behaviours

> **The mathematics. Monotonicity (`consistent_monotone` in `LowentRC11Mono.v`)**
>
> The same execution with only memory orderings weakened is also consistent — for all executions · all rf · all mo · all assignments. That is, **writing weaker orderings adds possible behaviours (never removes them).** At the edge level too, weakening removes synchronisation edges and adds none (`sw_weaken`). With `sc_is_the_strongest` and `rlx_is_the_weakest`, the strength order itself is a theorem.

Turn it around and it becomes a practical sentence — **writing a stronger memory ordering is always the safe direction.** That sentence is the ground for making `seq_cst` the default, and it is a theorem, not a belief.

> **In practice. A defect in the order found by the monotonicity proof**
>
> The old strength order placed `acq_rel` **above** `seq_cst`, because it counted only synchronisation edges and ignored the sequential consistency axiom. Yet store buffering is consistent under `acq_rel` and inconsistent under `seq_cst`. Under that order monotonicity was **false** (`old_order_is_not_a_strength_order` computes it). Why bounded exhaustive computation missed it was clear too — those skeletons never touched the sequential consistency axis. The order was fixed, making `seq_cst` the unique top.

46.8 Seven tests from the literature

Using only hand-picked executions cannot catch missing axioms (the failure story of E3). So the named tests from the literature were fed in (`LowentRC11Sweep.v`).

Test	`seq_cst`	`relaxed`
SB · LB · MP · 2+2W · IRIW	Forbidden	Allowed
CoRR · CoWR	Forbidden	**Forbidden**

Table 46.2 — Verdicts on seven litmus tests

The last line is the lesson. **"`relaxed` = no guarantee at all" is false.** The sequential consistency axis forbids store buffering and the coherence axis forbids CoRR and CoWR, and the latter is always there regardless of memory ordering.

> **Q. What happens to an atomic operation with no `order` written?**
>
> **A.** It is `seq_cst`. Code with no marks is the safest. Making `relaxed` the default would make code with nothing written dangerous, gaining danger **without asking for it**. Danger should be obtained by writing it. And combinations meaningless for an operation — `release` on a read, `acquire` on a write — are rejected at translation (chapter 27).

> **A common misconception. When performance matters, just write every atomic operation as `relaxed`**
>
> As E2 shows, `relaxed` **really permits** the outcome where both read 0. Removing constraints can make it faster, but what is removed is exactly the guarantee that you may reason sequentially. In this edition the only code with explicit weak ordering that has a proof is `spsc` (chapter 47); everything else is subject to audit. Write with the default `seq_cst`, measure, then pick the one place that must be weakened and write down the argument for that place.

46.9 What is not proven

- **A program logic verifying a particular lock-free algorithm under `relaxed`** is not in this chapter. The metatheorems within the model (sequential consistency · monotonicity · order

extension) are proven, and the only program verified with a program logic is the SPSC ring buffer, with a borrowed proof (chapter 47).

- **"If acyclic, topological sorting always succeeds"** is not there yet. For now the order is actually obtained and known correct, and failure shows as `None` — it never silently gives a wrong order.
- **The model is RC11 transcribed by hand.** Whether it matches the paper's definitions was checked by people reading, and it was wrong once (E3).
- **The tests are a sample.** No test uses fences (this model has no fence events), and each skeleton has one rf and mo — the bad outcome.
- **Whether the compiler keeps this model is separate.** Memory orderings are emitted one-to-one as C11 atomics, so the rest is the C compiler's job (the trusted base). That mapping and "atomic operations are not reordered" are confirmed by tests.

Recap

Sequential consistency means all accesses can be lined up in one row, and store buffering is its standard counterexample. RC11 views an execution as a graph with four kinds of edges and judges that an execution with a cycle does not exist. If every access is `seq_cst` the counterexample is impossible, with `relaxed` it is possible, and what forbids it is the sequential consistency axiom. `release` and `acquire` really synchronise message passing, and such bad executions contain races, so they have nothing to do with levels 1 and 2. For all executions `seq_cst` is sequentially consistent, and weaker orderings add behaviours, so writing stronger is always the safe direction.

47 Proofs about locks — where heavy tools are really needed

In this chapter

47.1 The vocabulary of separation logic 520
47.2 There cannot be two tokens 521
47.3 Tools only as needed 522
47.4 rwlock — two proofs of the same sentence 522
47.5 Deadlock freedom and no starvation 522
47.6 A borrowed proof in weak memory — `spsc` 523
47.7 What is not proven 525

What to know first

chapter 26, Tasks and channels · shared lock state does not exist in this edition yet (`E-LOCK-NOTYET`)

chapter 34, Containers and sorting · `spsc` is a lock-free single-producer single-consumer ring buffer

chapter 46, Proofs about weak memory · `release` and `acquire` synchronise message passing

Looking back

Where did chapter 45 say heavy concurrency logic is needed, and why only there?

> **A.** At level 3, and in particular for locks. The discipline of levels 1 and 2 removes two flows touching the same place altogether, so set theory was enough. But a spinlock is a device where two flows **really do hammer the same word at the same time**. Here discipline cannot give safety; an invariant must. This chapter covers that reasoning — concurrent separation logic.

The need for this chapter, and its context

The design decided that "lock and rwlock are libraries over atomic CAS". For that sentence to be true, a spinlock built from one CAS must really give mutual exclusion. And to trust the judgement of using tools only as much as needed, you must understand why locks alone use Iris while most proof files in this repository are pure Coq. This chapter introduces the vocabulary of separation logic from scratch, reads the theorems about locks, rwlocks, deadlock and starvation, and sees how a **borrowed** proof in weak memory backs the standard library's `spsc`. Locks are also the topic with the most boundaries in this part.

By the end of this chapter

You will learn the connective `∗` meaning "separate pieces", Hoare triples for writing specifications, invariants shared by several flows, and ghost tokens that exist only in the proof, not at run time. Starting from the theorem that there cannot be two tokens, you will pick up what the specifications for creating, acquiring and releasing a lock stop. You will also see the two proofs for rwlocks, deadlock freedom with ascending lock order and no starvation under round robin, and the borrowing of a weak-memory program logic to confirm `spsc`, together with its price.

The questions this chapter answers

1 If there is a proven lock, why is there no `lock` in this edition?

47.1 The vocabulary of separation logic

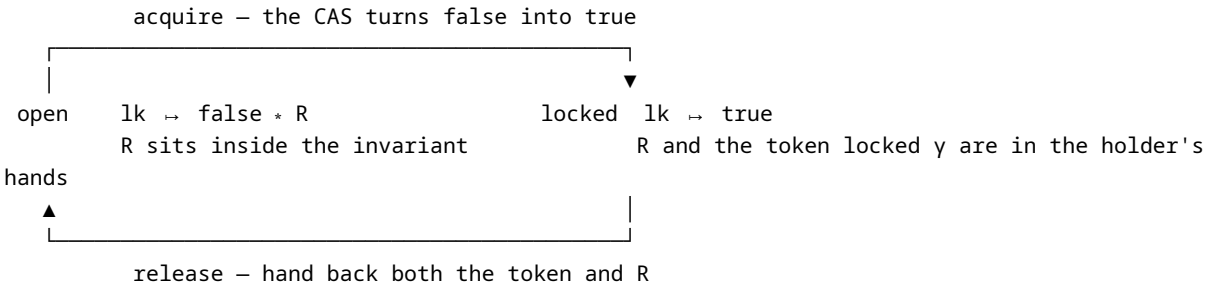
In ordinary logic, `A ∧ B` is "both are true". Separation logic adds one more connective. `A ∗ B` reads "A and B are true, and they are about **separate pieces of memory**". Claims about memory must imply **ownership**. `x ↦ 3` does not mean "x holds 3" but "I own x and its value is 3". So `x ↦ 3 ∗ y ↦ 4` says x and y are different places, and writing x does not spoil the claim about y. That is local reasoning.

It is the same idea as chapter 12's exclusivity rule. Actually the order is the reverse — Rust's borrow checker moved the idea of separation logic into a type system, and this language's exclusivity rule is in that lineage. chapter 42 is this chapter's idea brought down to translation time.

Term	Meaning
Hoare triple `{{{ P }}} e {{{ Q }}}`	Running `e` from a state satisfying `P` makes `Q` hold when it ends
Invariant `lock_inv γ lk R`	`(lk ↦ false ∗ R) ∨ (lk ↦ true)` — if open, resource `R` is inside the invariant; if locked, it is in the locking flow's hands
Ghost token `locked γ`	`own γ (Excl ())` — an accounting device absent at run time, circulating only in the proof

Table 47.1 — The vocabulary of lock proofs

The one-line invariant is the whole lock. `R` cannot be in two places at once, so only one flow holds `R`.



47.2 There cannot be two tokens

> **The mathematics. Theorems in `LowentLock.v`**
>
> `locked_exclusive : locked γ -∗ locked γ -∗ False.` Assuming two tokens is a contradiction. This is the mechanical heart of mutual exclusion.
>
> `newlock_spec : {{{ R }}} newlock #() {{{ lk γ, RET lk; is_lock γ lk R }}}.` **Hand over** the resource to protect and you get a lock. Making the lock costs giving up the resource — now the only way to reach it is to acquire the lock.
>
> `acquire_spec : {{{ is_lock γ lk R }}} acquire lk {{{ RET #(); locked γ ∗ R }}}.` Acquiring gives **both** the token and the resource. `∗` does the work — the resource received is mine and overlaps with no one's, so afterwards you reason as if there were one flow.
>
> `release_spec : {{{ is_lock γ lk R ∗ locked γ ∗ R }}} release lk {{{ RET #(); True }}}.` To release you must return both token and resource, and nothing remains afterwards.

If two flows acquired at once there would be two tokens, which is false. One release specification stops two defects. Releasing without acquiring lacks `locked γ` and cannot meet the precondition; using the resource after releasing fails because `R` was already returned.

The proofs share a three-step shape: open the invariant, run the CAS (splitting into success and failure), close the invariant. That the invariant may be opened **only at an atomic moment** is Iris's rule, and thanks to it "no other flow can see the invariant briefly broken" is guaranteed.

> **The mathematics. A practical example — two flows incrementing (`incr_spec`, `two_threads_spec`)**
>
> Two flows run `acquire ; c ← !c + 1 ; release` at the same time. Even incrementing concurrently they finish without races and the program yields a value. A lock-free `c ← !c + 1` — the

classic lost update — cannot be proven, because `c ↦ n` can be used only after receiving it from the lock. Honestly: this theorem does not say the result is **exactly 2**. Saying the exact total needs more ghost counting. The goal of this file was that the lock protects the resource.

47.3 Tools only as needed

Files	Tool	Why
Borrowing · loops · races · parallelism · RC11 · blocks · places · laundering · numbers	Pure Coq	Discipline, set theory, lattices and graph decisions suffice
Locks · rwlock read side	Iris	Two flows really hammer the same place
Weak memory programs · SPSC	iRC11 · gpfsl (borrowed)	Programs must be verified over weak memory

Table 47.2 — Proof files and tools

This is also why Coq was chosen. Iris is Coq-only and Lean has no mature counterpart. **Where the tools that would one day be needed lived** decided the proof language.

47.4 rwlock — two proofs of the same sentence

For the classic rwlock packing three meanings into one word — 0 empty, n > 0 n readers, −1 write-locked — **exclusivity of the write lock** was proven directly (`wlocked_exclusive`, `wlock_spec`, `wunlock_spec` in `LowentRWLock.v`). Because CAS succeeds **only from 0**, "a writer enters while readers are present" is stopped. Most real rwlock defects are exactly that.

Fractional ownership on the read side was already in Iris. Its rwlock interface is written with a fractional predicate `ϕ : Qp → iProp`, and `rw_spin_lock` is a verified instance. Readers get a fraction `ϕ q` (several at once), writing and reading cannot coexist, and a writer gets all of `ϕ 1`. What is valuable is that "writing and reading are exclusive" has **two proofs of the same sentence**. Ours shows it by "CAS succeeds only from 0" (close to the implementation); the borrowed one by ghost state (general, and covering fractions). The principle of building two ways and cross-checking (chapter 31) holds once more at the proof level.

47.5 Deadlock freedom and no starvation

The mathematics. Deadlock freedom (`no_deadlock` in `LowentDeadlock.v`)

If every flow takes locks **only in ascending order**, some flow can always progress in any state — they are never all blocked at once. The seed is one sentence. Look at **the flow wanting the largest lock**. If it is blocked, another flow holds that lock, and by the discipline what that flow wants is larger still — contradicting maximality. That breaking the discipline really deadlocks was also shown by computing a state (`violating_the_order_deadlocks`).

```
ascending order kept (A < B)          discipline broken
flow 1   holds A → wants B            flow 1   holds A → wants B
flow 2   wants A (waits)              flow 2   holds B → wants A
→ flow 1 takes B and goes on          → each waits for the other (deadlock)
```

Honestly — **the tool does not enforce this discipline.** So it is “written this way it is safe”, not “the compiler stops it”. Making it a check would require static orders on locks. And the lock’s own specification (acquiring gives the resource) does not prevent deadlock, because it says “acquiring gives” and not “it will be acquired”. Acquire a lock twice and release once, and the second acquire never arrives.

> **The mathematics. No starvation (`no_starvation` in `LowentFair.v`)**
>
> Under a cooperative round-robin scheduler a ready task always runs **within the number ahead of it + 1 steps**. Not “eventually” but bounded. That the induction measure is **position**, not queue length, is the point — rotation preserves length. That without rotation tasks starve was proven too.

The limits are clear. Cooperative scheduling **assumes yielding.** Before a task looping forever without `yield`, the scheduler can do nothing. Priorities and blocking are outside this model too.

> **A common misconception. Using locks also protects you from deadlock**
>
> A lock’s specification is “if you acquire it, you receive the resource”, not “it will always be acquired”. Deadlock freedom (`no_deadlock`) holds only under the discipline that every flow takes locks **in ascending order only**, and this edition’s tool does not enforce that discipline. Take a lock twice and release it once, and the second acquisition never comes. What the proof gives is a design rule, “used this way it is safe”; keeping that rule is still a person’s job.

47.6 A borrowed proof in weak memory — `spsc`

Iris’s language (`heap_lang`) is a sequentially consistent model. Proving code that writes weak memory orderings needs iRC11 and gpfsl, the Iris logics over RC11. `LowentIRC11.v` built the first theorem on them — in a message-passing program written with a `release` write and an `acquire` read, the reader always sees the data **under RC11**. It does not assume sequential consistency. chapter 46’s E4 rose from one execution to **a whole program**.

With that tool one standard library module was built.

examples/ch47/ring.low

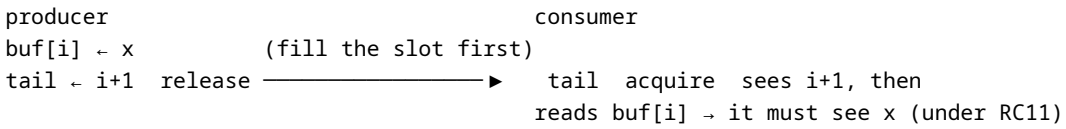
```
module ring .
rem run: main

use spsc .

proc main input k cap atomic . input al cap allocator . output u8 . effects atomic alloc
 .
do
  let gc option mut slice u8 be alloc_bytes al capacity 16 .
  let gb option mut slice u8 be alloc_bytes al capacity 32 .
  let go option mut slice u8 be alloc_bytes al capacity 8 .
```

```
  guard is_some gc . else return 255 .
  guard is_some gb . else return 254 .
  guard is_some go . else return 253 .
  var ctl mut slice u64 be view_array u64 (some_value gc) .
  var buf mut slice u64 be view_array u64 (some_value gb) .
  var out mut slice u64 be view_array u64 (some_value go) .
  let a u64 be spsc.spsc_push k ctl buf 10 .
  let b u64 be spsc.spsc_push k ctl buf 32 .
  let got u64 be spsc.spsc_pop k ctl buf out .
  guard eq got 1 . else return 252 .
  let first u64 be index out 0 .
  let again u64 be spsc.spsc_pop k ctl buf out .
  return narrow u8 (add first (index out 0)) .
end
```

Output

```
$ lowentc --run main ring.low
main() = 42
```

The producer **first** fills the slot and then publishes the index with `release`. The consumer reads that index with `acquire` before reading the slot. `lowent_spsc_push_is_correct` and `lowent_spsc_pop_is_correct` say this pair is correct under RC11 (including full and empty checks).



In practice. What “borrowed” means

The algorithm-level proof is gpfsl’s `circ_buff`, and `mp_instance_gen_inv` is gpfsl’s too. What this repository did was state and connect “the memory orderings we emit are exactly what that specification requires”. So gpfsl and the Iris development version join the trusted base. There is one reason not to rebuild it — keep two copies of the same thing and they diverge. And that is why **only one** was added. MPSC, MPMC and seqlocks have no grounds to borrow, and adding them without grounds would turn the standard library into a warehouse of “written as verified but verified by no one”. This work, once noted as “heavy”, turned out on measuring to be mostly assembling tools with no real risk. Before measuring, you do not know the cost.

Q. If there is a proven lock, why is there no `lock` in this edition?

A. What was proven is a CAS spinlock written in Iris’s language, and that the real runtime’s lock matches it is not verified — the largest gap between this file and the implementation. And the shared lock state type has not been built yet (`E-LOCK-NOTYET`, chapter 26). Here proof comes first and implementation later. Instead of accepting what does not exist and building it later, it says it does not exist.

47.7 What is not proven

- **The lock model is sequentially consistent.** Weak memory orderings are not in the `heap_lang` lock proofs. But since the default ordering is `seq_cst` (chapter 46), level 3 code written with defaults is inside that model, and only code writing weak orderings is left to iRC11. Containment holds here too.
- **There is no guarantee the implementation is this spinlock.**
- **`two_threads_spec` does not state the exact total.**
- **Deadlock freedom relies on a discipline people keep.** No starvation assumes yielding; priorities and blocking are outside.
- **There is no guarantee our compiler emits that `spsc` program.** One-to-one emission of memory orderings and not reordering them are confirmed by tests.
- **Borrowed proofs enter the trusted base.** Two checkers (Coq 8.20.1 · Rocq 9.2) giving the same answer reduces the chance of checker defects, but the two files needing gpfsl are confirmed only under Rocq 9.2.

Recap

Separation logic's $*$ means separate pieces and gives local reasoning, and a lock is proven with one invariant — "if open the resource is inside, if locked it is in the locking flow's hands" — plus a ghost token. There cannot be two tokens, acquiring gives token and resource, and releasing returns both. The rwlock's exclusivity was confirmed by a direct proof and a borrowed one, ascending lock order does not deadlock, and round robin runs tasks within a bound. `spsc` was confirmed by borrowing a weak-memory program logic, at the price of a larger trusted base.

48 Proofs about syntax — whichever closer closes it, the same tree

In this chapter

48.1 Algebra and denotation 526
48.2 One statement is a one-statement block 527
48.3 Closers do the same job 528
48.4 Common mistakes 530
48.5 What is not proven 531

What to know first

chapter 3, The surface · a detached period closes, and a newline is whitespace
chapter 8, Expressions · prefix notation has no precedence
chapter 39, The mathematical toolkit · sets and relations — what decides "the same"

Looking back

In chapter 3, `poly`'s `return` spanned two lines but was one form. What closed that form, and what did the newline do?

> **A.** The detached period at the end of the second line closed it. A newline is whitespace exactly like a space, so it closed nothing. That is why splitting the line anywhere kept the meaning. This chapter covers why such surface properties are subjects of proof.

The need for this chapter, and its context

The proofs so far were about **meaning** — what a program does. This chapter is about **surface** — what a program looks like. Syntax needs proofs because of something that actually happened. The parser was chopping forms at `.` and a later stage was silently gluing them back — two layers believed different grammars, and the difference came out as defects. For the principle "one spelling per meaning" not to remain a matter of surface taste, a machine must guard that two spellings build the same tree.

By the end of this chapter

You will learn monoids, which have an identity and associativity, and denotations, which decide by a function whether two spellings are the same. You will confirm with examples the theorem that one statement equals a one-statement block, the theorem that the three closers build the same tree, and associativity of blocks. You will also see newline closing, left out even though its equivalence is provable, the theorems removed along with removed rules, and what these theorems do not say about the real parser.

The questions this chapter answers

1. With this theorem, can the real parser be trusted to be correct?
2. C has dozens of lines of precedence rules; why are this chapter's proofs so short?

48.1 Algebra and denotation

Joining statements has two properties. The way they join does not depend on grouping (associativity), and an empty block is nothing (identity — what 0 does for addition). A structure with both is a **monoid**. Concatenating lists is the typical monoid, and a program's sequence of statements has exactly that structure.

To say whether two spellings are "the same" you need a criterion. Take a function `denote` sending each tree to **a flat sequence of statements**, and call them the same if the results match. `denote(do S end) = [S] = denote(S)`, so the two are the same. This is the only technique in this chapter — define "the same" by a function and confirm equations by computation.

48.2 One statement is a one-statement block

examples/ch48/oneform.low

```
module oneform .
rem run: bare 3
rem run: blocked 3
rem fmt

fn bare input a u64 . output u64 .
  requires le a 1000 .
do
  var x u64 be a .
  if gt a 1 . set x (add x 10) .
  return x .
end

fn blocked input a u64 . output u64 .
  requires le a 1000 .
do
  var x u64 be a .
  if gt a 1 . do
    set x (add x 10) .
  end
  return x .
end
```

Output

```
$ lowentc --run bare oneform.low 3
bare(3) = 13
$ lowentc --run blocked oneform.low 3
blocked(3) = 13
$ lowentc --fmt oneform.low
module oneform .
fn bare .
  input a u64 .
  output u64 .
  requires le a 1000 .
do
  var x u64 be a .
  if (gt a 1) do set x (add x 10) . end
  return x .
end
fn blocked .
  input a u64 .
  output u64 .
  requires le a 1000 .
do
  var x u64 be a .
  if (gt a 1) do set x (add x 10) . end
  return x .
end
```

`bare` writes one statement directly as the body of `if`, and `blocked` wraps it in `do … end`. The answers are the same, and the canonical form printed by `--fmt` does not differ by a single character.

> **The mathematics. Theorem G1 (`one_form_is_a_block` in `LowentBlock.v`)**
>
> `forall s, denote (wrap s) = denote s.` Wrapping one statement in a block does not change its meaning. Here "same meaning" means that the function `denote`, sending each tree to a flat sequence of statements, gives the same result. Defining "the same" by a function and confirming equations by computation is the only technique in this chapter. The corollary `body_is_one_thing` says that writing a statement or a block in a body position is the same.

This theorem justifies a design decision. C separates statements and compound statements into different kinds. `if (c) x = 1;` and `if (c) { x = 1; }` are grammatically different, and the famous `goto fail;` trap came from that gap — the indentation made two lines look as if they belonged to the `if`, but the grammar made only one belong. Lowent does not separate them.

```
C       if (c)
            x = x + 10;
            y = 0;          ← indented as if inside the if, but the grammar puts it outside

Lowent  if gt a 1 . set x (add x 10) .        one statement = a one-statement block (G1)
        if gt a 1 . do
          set x (add x 10) .
          set y 0 .
        end                                   a longer body is ended by end
```

48.3 Closers do the same job

> **The mathematics. Theorem G2 (`closers_agree`)**
>
> `forall c1 c2 h ops, close c1 h ops = close c2 h ops.` Closing with `.` or `)` gives the same tree. Both "close the innermost open form". `end` is not on the list — `do … end` is a pair of braces, so `end` closes only its own `do`; if a form is still open inside, it does not close it for you but refuses (`E-DOT-MISSING`). Blocks are associative, so a block within a block unfolds in place (`block_assoc`, `nested_block_flattens`).

Seen in an example:

examples/ch48/closers.low

```
module closers .
rem run: one_line 4
rem run: spread 4

fn one_line input x u64 . output u64 .
  requires le x 1000 .
do
  return add (mul x x) (add x 1) .
end

rem spreading arguments over several lines needs no marker --- a newline is whitespace
fn spread input x u64 . output u64 .
```

```
  requires le x 1000 .
do
  return add
    (mul x x)
    (add x
         1) .
end
```

Output

```
$ lowentc --run one_line closers.low 4
one_line(4) = 21
$ lowentc --run spread closers.low 4
spread(4) = 21
```

In `one_line` the two `)` close the inner forms and `.` closes the outer one. `spread` lays the same form over four lines with no marks needed, because a newline is whitespace. Both ops give the same answer.

```
return add (mul x x) (add x 1) .      return add
                                          (mul x x)
                                          (add x
                                               1) .

both become the same tree:
                 add
            ┌────┴────┐
           mul        add
          ┌─┴─┐      ┌─┴─┐
          x   x      x   1
```

A newline could also be made a closer, and that is provable too. It still is not done. If newlines closed, line breaks would carry meaning, and merely splitting a long line would change the program. **That a meaning can be proven the same does not mean the spelling must exist.** Theorems once attached to commas, line continuation and newline closing were all `Qed` too, but when those rules left the language, the theorems left with them. Today `,` and `;` are rejected with `E-VOCAB-REMOVED` — a word that parses but has no effect is a silent trap. Keeping theorems modelling dead rules would make a proven document speak of rules that do not exist.

Every proof in this file ends by computation. The value is not in the difficulty of the proofs but in **having chosen and written the propositions**. With "closers do the same job" written down, the proof breaks when someone treats one specially. Syntax is a layer that quietly drifts with "it would be convenient to make an exception just here", and theorems catch that drift. What they stop are not users' defects but the compiler's.

> **Q. With this theorem, can the real parser be trusted to be correct?**
>
> **A.** No. The model is a few small functions building trees, and the real parser is much larger. Their correspondence is backed by tests. And the back-end cross-check compares the VM and native code, but both pass through **the same front end**, so if the parser is wrong both are wrong in the same way. An attempt to cross-check the parser directly against this model did not get enough pairs to conclude anything, because the model is smaller than the real grammar. The front end is the weakest seam in this language's verification (chapter 50).

Q. C has dozens of lines of precedence rules; why are this chapter's proofs so short?

A. This language's expressions are prefix, so there is no operator precedence (the `expr` island is handled separately, chapter 8). With no precedence there are few cases of "what tree does this spelling become" to answer. It is a case of design making proofs easy. But short proofs are not small in value — having written the propositions down is itself what prevents drift.

A common misconception. A line break ends a statement

A line break is whitespace. `spread` in `closers.low` lays one form over four lines and, without any mark, becomes the same tree as `one_line`. Only closers (`.·)`) end a form. So a missing stop makes the next line join the previous form and produces an odd diagnostic, while splitting a long expression over several lines never changes its meaning. A rule making line breaks closers can also be proved, and this second property is exactly why it was not adopted.

48.4 Common mistakes

Counter-example. Leaving out the period at the end of a line

examples/ch48/mistake_noperiod.low

```
module mistake_noperiod .
rem expect: E-IR-ARITY

fn total input a u64 . input b u64 . output u64 .
  requires le a 1000 .
  requires le b 1000 .
do
  rem ✘ the period at the end of this line is missing
  var s u64 be add a b
  set s (mul s 2) .
  return s .
end
```

Output

```
$ lowentc --check mistake_noperiod.low
mistake_noperiod.low:10:0 E-IR-ARITY: extra operands in expression (strict arity)
```

The diagnostic points not at the line missing its period but at **the next line**. A newline closes nothing, so `set s (mul s 2)` was joined onto `add a b`, `add` ended up with four operands, and the tool says "extra operands". Look one line above the reported number first. Even if this diagnostic is confusing, the rule stays. If a line ended a statement, merely splitting a long expression would change its meaning.

Counter-example. Closing with ; as in C

examples/ch48/mistake_semicolon.low

```
module mistake_semicolon .
rem expect: E-VOCAB-REMOVED
```

```
fn twice input a u64 . output u64 .
  requires le a 1000 .
do
  rem ✘ closing with a semicolon, as in C
  return mul a 2 . ;
end
```

Output

```
$ lowentc --check mistake_semicolon.low
7:20 E-VOCAB-REMOVED: `;` was a THIRD spelling of the closer `.` (the lexer literally em
itted a DOT for it) — one meaning, three spellings, and SPEC-002 §2.5 forbids synonyms.
It only survived inside the old interpreter's second language. Write `.`
```

`;` used to be a third spelling of the period. Three spellings for one meaning make readers hunt for a difference, so it was removed and is now refused with `E-VOCAB-REMOVED`. It is not silently accepted, because a word that parses but means nothing is the hardest trap to find.

48.5 What is not proven

- **There is no guarantee that the real parser is this model.** The model is a few small functions building trees, and the real parser is much larger. The correspondence is backed by unit and regression tests.
- **`denote` is a flat sequence.** Position information, comments and error recovery were not modelled. So “the same tree” means the same structure, not the same diagnostic quality.
- **The lexical (token) layer is separate.** Whether tokens split exactly into this model’s three closers is the lexer tests’ job.

Recap

A sequence of statements is a monoid with associativity and identity, and whether two spellings are the same is decided by a denotation into flat sequences, confirmed by computation. One statement equals a one-statement block (G1), `.` and `)` build the same tree (G2), and a block within a block unfolds in place. Newline closing was left out even though provable, and theorems of removed rules were removed with them. These theorems stop the compiler from drifting, but that the real parser is this model is backed by tests.

49 Proofs about hashes — calling things by content, not name

In this chapter

49.1 Hash functions and Merkle DAGs 532
49.2 Called by content, not by name 533
49.3 What changes a hash and what does not 533
49.4 What can and cannot be proven 535
49.5 What hashes stop 536

49.6 What is not proven 537

What to know first

chapter 31, Building and testing · builds are cached by the content hash of emitted C, and dependencies are pinned by hash
chapter 3, The surface · the clause order of a head is fixed to one
chapter 48, Proofs about syntax · two spellings with the same meaning become the same tree

Looking back

In chapter 31, what did `--lock` refuse to build over even when the version number is the same?

> **A.** Different bytes. Even with the same version number, changed content is a different dependency. This chapter covers how to write that "content" so that **the same program** is recognised by meaning rather than bytes, and what can be proven about how it is written.

The need for this chapter, and its context

If chapter 48 was about two spellings building the same tree, this chapter is about whether two programs **are the same**. If two programs a cache judges "the same" differ in meaning, the cache lies. Conversely, judging programs with the same meaning "different" rebuilds needlessly. Reproducible builds, incremental builds and dependency pinning all rest on this judgement. The hash function itself is an unprovable cryptographic assumption, but what comes **before** it — what goes into the hash and what is left out — can be proven.

By the end of this chapter

You will learn the assumption of collision resistance, the Merkle DAG in which hashes contain hashes, and how ops calling each other are grouped into one bundle to remove cycles. You will check against real output how the two hashes of `--emit-db` (`iface` and `def`) react to comments, bodies, effect order and contracts, and understand what the injectivity theorem of the canonical encoding guarantees and why the imprecision of not normalising contract clauses is the safe direction.

The questions this chapter answers

1 Are `--emit-db` hashes actually used for build decisions?

49.1 Hash functions and Merkle DAGs

The hash function BLAKE3-256 reduces bytes of any length to 32 bytes. One property is needed — **no pair of different inputs giving the same output can be found** (collision resistance). So "the hashes are equal" may be used as "the contents are equal". This is not proven mathematically but an **assumption**, and it enters the trusted base (chapter 50). Still, in practice it is sturdier than any other assumption.

49.2 Called by content, not by name

Ordinary build systems judge caches by file names and modification times. So a file touched with unchanged content is rebuilt needlessly (a loss), and changed content with the same time uses a stale result (a danger). **Content addressing** drops names and calls things by the hash of their content. `lowentc --emit-db` gives two hashes per op — `iface`, the surface observable from outside (signature and contracts), and `def`, the whole definition. The hash function is BLAKE3-256.

A hash embeds its dependencies **by their hashes**. Then a definition's hash reflects the content of everything it depends on, and when one op deep down changes, the hashes above change along with it. The hash itself is dependency tracking. This is a Merkle DAG.

Ops calling each other create cycles. If `hash(A)` contains `hash(B)` and `hash(B)` contains `hash(A)`, the definition goes round. So the cyclic ones are grouped into **one bundle** (a strongly connected component, SCC). A nameless preliminary hash fixes the canonical order within the bundle, calls inside the bundle carry that order's number instead of a name, the bundle is hashed once, and each member's hash is (bundle hash, its number). Contracting strongly connected components leaves an acyclic graph — the same idea as chapter 39's fixed points appears again.

```
A ⇄ B (they call each other)      C ──▶ A

1  group {A, B} into one block (an SCC)
2  order the block with nameless preliminary hashes        A = 0 · B = 1
3  calls inside the block carry the number, not the name   A calls B → "number 1"
4  hash the block once                                     → H
5  hash each member as (H, its number)                     hash(A) = h(H, 0) · hash(B) = h(H, 1)
   C carries hash(A) — the graph that is left has no cycles
```

49.3 What changes a hash and what does not

examples/ch49/h1_plain.low

```
module hashed .
rem db

export fn twice input a u64 . output u64 .
do
  return add a a .
end
```

Output

```
$ lowentc --emit-db h1_plain.low
fn twice/1 iface:6630705cb68be394 def:73e3a88345444202
```

examples/ch49/h1_comment.low

```
module hashed .
rem db

rem adding a comment leaves the hash unchanged
export fn twice input a u64 . output u64 .
```

```
do
  rem a comment here too
  return add a a .
end
```

Output

```
$ lowentc --emit-db h1_comment.low
fn twice/1 iface:6630705cb68be394 def:73e3a88345444202
```

H1 — comments change nothing. Both hashes are the same. Fixing a comment does not rebuild the whole project. And a machine keeps the definition of "the same program" as meaning, not bytes.

examples/ch49/h2_body.low

```
module hashed .
rem db

export fn twice input a u64 . output u64 .
do
  return mul a 2 .
end
```

Output

```
$ lowentc --emit-db h2_body.low
fn twice/1 iface:6630705cb68be394 def:64a7865b02486293
```

H2 — changing only the body changes only `def`. Changing `add a a` to `mul a 2` left `iface` as it was and changed only `def`. This is the heart of incremental builds. When only an op's body changes, its users are recompiled, but their `iface` stays the same too, so the next dependants hit the cache. Signature changes spread along the chain; body-only changes stop after one step.

```
main ──calls──▶ util ──calls──▶ leaf        only leaf's body changed

leaf   def changes · iface stays
util   recompiled · its iface stays too
main   cache hit — it stops here
```

examples/ch49/h3_effects_a.low

```
module hashed .
rem db

proc greet input out cap io . input al cap allocator . output u64 . effects io alloc .
do
  let g option mut slice u8 be alloc_bytes al capacity 4 .
  return write_out out 1 "hi" .
end
```

Output

```
$ lowentc --emit-db h3_effects_a.low
proc greet/2 iface:d864e930303f7437 def:bebc1fcf6231d282
```

examples/ch49/h3_effects_b.low

```
module hashed .
rem db

proc greet input out cap io . input al cap allocator . output u64 . effects alloc io .
do
  let g option mut slice u8 be alloc_bytes al capacity 4 .
  return write_out out 1 "hi" .
end
```

Output

```
$ lowentc --emit-db h3_effects_b.low
proc greet/2 iface:d864e930303f7437 def:bebc1fcf6231d282
```

H3 — effects are a set. `effects io alloc` and `effects alloc io` give the same hash. They once differed, because the clause's source text was hashed. Then content addressing addressed **spelling**, not meaning. The grammar says effects are a set, so they are hashed in normalised order.

examples/ch49/h4_contract.low

```
module hashed .
rem db

export fn twice input a u64 . output u64 .
  requires le a 100 .
do
  return add a a .
end
```

Output

```
$ lowentc --emit-db h4_contract.low
fn twice/1 iface:31247d9015a6c804 def:a088ec347ad280ef
```

Contracts are surface too. Adding `requires le a 100 .` changed `iface`. Callers trust and check the contracts of the ops they call, and as seen in chapter 41, contracts are the grounds for removing bounds checks. If a contract changed but the hash stayed the same, the cache would keep using results that trusted the old contract — like reusing a proof after its premises changed. The implementation once hashed only signatures and had this hole; it was fixed.

49.4 What can and cannot be proven

`LowentHash.v` separates the two exactly. BLAKE3's collision resistance is a cryptographic assumption, not a subject of proof. What can be proven comes **before** it — what the canonical encoding erases and what it keeps.

> **The mathematics. Encoding theorems (`LowentHash.v`)**
>
> `H1_comments_are_irrelevant` · `H2_body_does_not_touch_iface` · `H3_effects_are_a_set`, and `encoding_is_injective` — if encodings are equal, every piece of the interface is equal. The last

> theorem is the valuable one. If “same hash but different interface” ever occurs, the cause narrows to one thing — a collision of the hash function. A machine guarantees it did not come from an ambiguous encoding.

Unlike effects, contract clauses are not normalised but hashed **in written order**. So `requires A . requires B .` and `requires B . requires A .` have the same meaning but different hashes. It is imprecise. But it is the safe direction. Same meaning with different hashes only runs one more rebuild, while different meanings with the same hash make the cache lie. That this direction is safe is proven too. The asymmetry seen in the interval analysis (chapter 41) and borrow checking (chapter 42) — **when unsure, go wide** — decided the judgement here too.

> **A common misconception. Leaving the order of head clauses free would make hashes unstable**
>
> That is why the order was fixed to one (chapter 3). Other orders are rejected at translation and fixed by `--fmt`, so the same program never has two hashes. The principle of one spelling per meaning is not a matter of surface taste but also a premise of content addressing.

49.5 What hashes stop

Situation	Stopped by
Fixing a comment rebuilds everything	H1
Fixing a body rebuilds the dependants of dependants	H2
Only reordering effects misses the cache	H3
Changing a contract lets dependants use old results	Contracts are in `iface`
A dependency silently changes	The Merkle DAG, since hashes contain content
A lock file’s dependency changes	The lock file’s content hash refuses it (chapter 31)

Table 49.1 — What content addressing stops

The last two lines are the grounds of reproducible builds. Making “the same source gives the same result” **comparable** is the value of content addressing — it stops being something believed and becomes something compared.

> **Q. Are `--emit-db` hashes actually used for build decisions?**
>
> **A.** Not fully yet. `--emit-db` is a side file, and this edition’s incremental builds run on the content hash of emitted C (chapter 31). The `iface` and `def` hashes are the layer that records exactly what is the same and what differs, and tests guard those same/different relations. Hashes taking over every build decision is future work.

49.6 What is not proven

- **The theorems are about a model.** `LowentHash.v` proves H1, H2 and H3 for the canonical encoding. That the implementation uses that encoding is confirmed by tests on a few examples.
- **Normalisation is incomplete.** It is sensitive to contract clause order, and how far local variable names or numeric literal spellings (`0x10` vs `16`) are normalised was not exhaustively checked.
- **BLAKE3's collision resistance is an assumption.** If wrong, the cache sees different contents as the same.
- **That bundle (SCC) hashes come out the same across implementations is a specification-level promise** and not settled. There are several ways to handle cycles.
- **What tests guard is not hash values but the relations same and different.** If the encoding changes, values change, but the relations must stay.

Recap

Collision resistance of the hash function is an assumption, a Merkle DAG in which hashes contain dependencies' hashes replaces dependency tracking, and cyclic ops are hashed as one bundle. `--emit-db`'s `iface` and `def` hashes do not react to comments; body changes change only `def`, contract changes change `iface` too, and effects are normalised as a set. The injectivity of the encoding is proven, so if hashes match but interfaces differ the cause narrows to a collision. The imprecision of not normalising contract clauses is the safe direction.

50 What is not proven

In this chapter

50.1 The trusted base — what proofs rely on 538
50.2 The largest gap — between model and implementation 539
50.3 What remains — by topic 540
50.4 Not sold as "fully statically safe" 542
50.5 So what may you trust 542

What to know first

chapter 38, Why prove · proven, exhaustively checked and sketch are not mixed
chapter 31, Building and testing · five ways of checking the compiler against something
chapter 48, Proofs about syntax and hashes · the front end is the weakest seam

Looking back

Why did chapter 38 call “a program written in a proven language is correct” a misconception?

> **A.** Because proofs are about **the language’s rules**, not **your program**. The theorem that the borrowing rules are sound says only that programs passing translation have no borrowing violations. This chapter follows that boundary to the end and gathers in one place what the proofs rely on and what they do not cover.

The need for this chapter, and its context

Documents boasting of proofs usually do one of two things — they do not write the boundaries, or they excuse themselves in one line at the front and forget. Both lead readers to false confidence. **The value of a proof comes from knowing what it does not cover.** That this chapter ends the book is no editorial accident. Only after knowing what the earlier chapters claimed do the boundaries mean anything. Quoting this book without reading this chapter makes it say things it did not say.

By the end of this chapter

You will learn the trusted base the proofs rely on (the Coq kernel, the C compiler, BLAKE3, borrowed proofs and so on) and how much collapses if each is wrong. You will separate the devices bridging the gap between model and implementation into empirical and static, and understand that the weakest seam is the front end. You will also sort out what remains by topic, why this language must not be called “fully statically safe”, and in the end what you may trust.

The questions this chapter answers

1 So what do all these proofs buy in the end?

50.1 The trusted base — what proofs rely on

Trusted	Why trusted	If wrong
The Coq/Rocq kernel	A small core, and two versions (8.20, 9.2) give the same answer	Every theorem becomes meaningless
Iris	Needed only for the lock proofs	Only the lock theorems collapse
iRC11 · gpfsl · Iris’s `rw_spin_lock`	Proofs of weak memory, SPSC and rwlock read side were borrowed	Only those theorems collapse
The C compiler (gcc, clang)	There is no alternative	Native code diverges from the intermediate representation

BLAKE3's collision resistance	A premise of content addressing	The cache sees different contents as the same
Operating system and hardware	There is no alternative	Everything
`proven_c_lib` (base library)	Has its own tests	Data structures go silently wrong
The model captures the language correctly	Confirmed by people reading it	The gap below

Table 50.1 — The trusted base

50.2 The largest gap — between model and implementation

```
The theorems are about a model.
The model was written by people.
The compiler is different code written in a different language.
That the two are the same is not proven.
```

These are the devices joining the seam.

Device	What it does	Character
Bounded exhaustive model checking	Whether model and implementation verdicts agree on every case of small size	Empirical
Events extracted from real programs	Feeds borrowing events of real programs to the same model	Empirical · real programs
Back-end cross-check	Whether the VM and native code give the same answer	Empirical
Analysis self-accusation (`E-VM-ANALYSIS`)	Refutes the compiler's "may be removed" by execution	Empirical
Certificate recheck	An independent checker re-examines the arithmetic grounds of removed checks	Static
Regression tests	Changed output becomes visible	Regressions only

Table 50.2 — Devices joining model and implementation

The first four are empirical. They say not "the output is correct now" but "these inputs did not disagree". The certificate recheck is different in character. It speaks even when no input hits that place. But it too **trusts the facts and checks only the reasoning**. The two layers catch different things. Cross-checks caught defects proofs missed, and proofs caught defects cross-checks missed. That is why they are overlapped.

The weakest seam is the front end (lexer and parser). The back-end cross-check compares the VM and native code, but both pass through the same front end. If the front end is wrong, both are wrong the same way. An attempt to cross-check the parser directly against the grammar model did not get enough pairs to conclude anything, because the model is smaller than the real

grammar. Closing this place requires a second independent implementation of the front end to compare against, and that cost (maintaining two) has not been paid yet.

> **In practice. Discrepancies this book found**
>
> Every example in this book was run on the VM and natively and compared, and rejected examples were checked down to their diagnostic codes. In the process, close to twenty discrepancies between the compiler and the specification came to light. Some are written in the text — the effects of ops called through `method` not spreading to the caller (chapter 23), `ensures` with an expression going unchecked without warning (chapter 14), `arg` numbering differing between the two back ends (chapter 2), and the precedence of comparisons and `and` in the `expr` island differing from the specification's table (chapter 8). All are discrepancies on the side of **implementation** and **documentation**, not the proofs' **model**. Without the discipline of actually running examples before printing them, this book would have written those discrepancies as facts.

50.3 What remains — by topic

What	Status
Effect system	Proven. The rule by which `task_group` absorbs `concurrent` and effect closure are outside the model
Effect-based optimisation	Proven (reordering, common subexpression elimination, memoisation, dead code elimination). Call boundaries and the moment of stopping are outside
Loop termination	Not covered. Infinite loops are legitimate programs
Soundness of the whole type system	Partial. There is no effect-row and no proof of it
Completeness of normalisation	Partial. Effects normalised as a set (proven). Contract clauses are sensitive to written order, proven to be the safe direction
Multi-implementation compatibility of bundle hashes	Unsettled. A specification-level promise
Parser ≡ grammar model	Not proven

Table 50.3 — Language core and tools

What	Status
General RC11 weak orderings	Partial. Metatheorems within the model are proven. Only SPSC (a borrowed proof) verifies a specific lock-free algorithm under weak orderings
Deadlock freedom	Proven. But the ascending lock order discipline must be kept by people (the tool does not enforce it)

No starvation	Proven. Assumes round-robin and yielding; priorities and blocking are outside
Implementation of level 1 parallelism	Implemented. The theorems do not verify the implementation
Lock-free data structures	One SPSC ring buffer. MPSC, MPMC and seqlocks have no proof to borrow, so they were left out
Atomic orderings → machine code	Emitted 1:1 as C11 atomics, so the rest is the C compiler's (trusted base) job. The mapping itself is confirmed by tests

Table 50.4 — Concurrency

What	Status
Floating-point properties (rounding, NaN, −0)	Not covered
Floating-point support in the interval analysis	None
Effect atoms such as `page_fault` and `blocking`	Vocabulary only, no primitive operations
Effect-row polymorphism	None

Table 50.5 — Feature scope

examples/ch50/floats.low

```
module float_gap .
rem run: tenths
rem run: close_enough

fn tenths output bool .
do
  return eq (add 0.1 0.2) 0.3 .
end

fn close_enough output bool .
do
  let d f64 be sub (add 0.1 0.2) 0.3 .
  return lt (abs d) 0.000000001 .
end
```

Output

```
$ lowentc --run tenths floats.low
tenths() = 0
$ lowentc --run close_enough floats.low
close_enough() = 1
```

0.1 plus 0.2 is not equal to 0.3. This is not a defect but a property of IEEE 754, and this language's proofs did not cover that property. Floating-point equality is asked with a tolerance, as in `close_enough`. In computations involving floating point, this book's "proven" guarantees nothing.

50.4 Not sold as “fully statically safe”

```
Stopped statically   borrowing · regions · reference escape · parallel overlap · integer widening
Stopped at run time  bounds checks where unproven · overflow · contract checks
                     dangling references of generational handles
```

This language is a **mixed** design. And it does not call that static. Written honestly as a comparison, Rust has proofs through RustBelt, and Zig does not claim safety (an honest stance too). Lowent is “has proofs — **mainly for the sequential fragment**”. Remove that qualifier and it becomes a lie. And the `release_fast` build mode removes the remaining contract checks, opening one more trust boundary (chapter 14). “No undefined behaviour” is true only within the safe subset.

> **Q. So what do all these proofs buy in the end?**
>
> **A.** Two things. First, **what the compiler must reject** is decided by argument, not feel. Whether one rule is enough, whether a condition is ≤ or <, come from proofs. Second, defects the implementation hit remain as **theorems**. Stories are forgotten, but if someone removes the check, the proof breaks. Proofs do not guarantee that programs are correct, but they systematically reduce the places where the language is silently wrong.

50.5 So what may you trust

```
✓ Integers never silently change value                  (numbers · proven)
✓ Code passing translation has no borrowing violations  (ownership · proven, sequential)
✓ Effect declarations cover effects that happen         (effects · proven, within the model)
✓ Safe code has no data races                           (concurrency · proven)
✓ Parallel results equal sequential, bit for bit        (parallel · proven)
✓ With the default ordering you may think sequentially  (weak memory · proven, ∀ executions)
✓ Removed bounds checks are at places argued in range   (bounds · proven + rechecked)

△ The compiler follows this model exactly               (empirical)
△ Code stating weak orderings                           (only SPSC proven — the rest is subject
to audit)
✘ Floating-point details                                (not covered)
```

You may trust documents that have these last lines. Documents without them need a second look.

> **A common misconception. A ✔ line means this edition’s compiler always behaves that way**
>
> A ✓ is a theorem about the **model**; whether the compiler follows the model belongs to the △ line — empirical. While writing this manual several of those gaps were measured. When a `reduce` accumulator does not start at the identity, the split native answer differs from the sequential one (chapter 27); lending a `let` name with `mut_ref` breaks its immutability (chapter 12); and passing a number where a capability belongs lets an op without the capability print (chapter 16). Each is written as a warning in its chapter and filed as a defect in the development repository. A ✓ is the reason something must be stopped; to know whether it is, read it together with that chapter’s warnings and the defect list.

Recap

The proofs rely on a trusted base such as the Coq kernel, the C compiler, BLAKE3, borrowed proofs and the operating system. The largest gap is between model and implementation, bridged empirically by exhaustive checking, cross-checks and self-accusation, and statically by certificate rechecks. The weakest seam is the front end. Loop termination, floating point, effect-rows and the parser's correspondence to the model are outside the proofs. This language mixes static and run-time checks, and its proofs are mainly about the sequential fragment. Read with what you may trust, what is empirical and what is not covered kept apart.

Appendices

Appendix A — Words and builtins

The text is where words are **explained**. This appendix gathers only tables to skim when you get stuck reading code. The word list is the same as the specification's Appendix A, and anything not in that list is not a word.

The forty-three words

Kind	Words	Chapter
Declarations	`module use type newtype struct enum trait contract actor state`	chapters 21 and 13
ops	`fn proc export unsafe extern satisfies`	chapter 5
Locals	`let var set be`	chapter 6
Flow	`if else while for guard match case return break continue try`	chapter 7
Expressions and values	`expr make true false none`	chapter 8
Concurrency	`spawn send`	chapter 25
Ownership	`drop`	chapter 19
Tests	`test expect`	chapter 31
Blocks	`do end`	chapter 3

Table 50.1 — All the words

`rem` and `note` are markers opening comments, and clause heads such as `input`, `output`, `effects` and `requires` and builtins such as `add` and `len` are not words but cannot be used as names.

Removed words

Removed	Instead
`loop`	`while true .`
`give`	`return`
`unit`	`void`
`calcop` · `procop`	`fn` · `proc`
`is` · `as` (in declarations) · `local`	Not written
`to` · `in` (field access)	`field a b` · `index a i`
`on`	`proc` inside an actor
`fail`	`return error <variant>`
`;` · `,`	`.`

Table 50.2 — Removed words and what to use instead (`E-VOCAB-REMOVED`)

`as` and `to` remain only as position markers in `use … as <alias>` and `case <low> to <high>`.

Frequently used builtins

Kind	Operations
Arithmetic (stopping)	`add` `sub` `mul` `div` `mod` `neg` `abs` `min` `max`
Arithmetic (choosing the outcome)	`wrap_*` `sat_*` `chk_*` `div_nz` `nonzero_of`
Changing width	`widen` `narrow` `narrow_wrap` `narrow_sat` `narrow_try` `cast` `bit_cast`
Comparison and logic	`eq` `ne` `lt` `le` `gt` `ge` `and` `or` `not`
Bits	`bit_and` `bit_or` `bit_xor` `bit_not` `shl` `shr` `rotl` `rotr` `wrap_shl` `wrap_shr` `count_ones` `leading_zeros` `trailing_zeros` `byte_swap` `clmul_lo` `clmul_hi`
Sequences	`len` `index` `subslice` `view` `view_array` `same_slice`
Groupings	`field` `method` `isa` `get`
Answer-carrying types	`some` `ok` `error` `is_some` `is_none` `is_ok` `is_error` `some_value` `ok_value` `error_value` `value_or`
Translation time	`size_of` `comptime` `config`
Allocation	`alloc_bytes`
I/O leaves	`write_out` `read_in` `arg` · file, connection, clock, random and terminal leaves are wrapped by the standard library
Atomic	`atomic_load` `atomic_store` `atomic_add` `atomic_sub` `atomic_swap` `atomic_cas` `atomic_fence`
Devices	`read_volatile` `write_volatile`
Concurrency	`await` `drain` `channel` `chsend` `chrecv` `yield` `spawn` `send`
Atomic (bits)	`atomic_and` `atomic_or` `atomic_xor`
Floating-point maths — chapter 4	`sqrt` `sin` `cos` `exp` `log` `pow` `floor` `ceil` `round` `fmod` `sum_neumaier` `sum_seq`
Pipe stages — chapter 24	`pipe` `map` `filter` `fold` `scan` `take` `skip` `zip` `enumerate` `reverse` `collect` `count` `all` `any` `into`
Lanes (SIMD) — chapter 27	`splat` `load` `store` `load_masked` `store_masked` `select` `reduce_add` `reduce_mul` `reduce_min` `reduce_max` `avg` `native_lanes` `rotate` `prefetch`
Small-number sets — chapter 13	`bitset_new` `union` `intersect` `difference` `complement` `contains` `is_subset` `is_empty` `remove`
Layouts and views — chapter 13	`encode` `try_view` `view_segments` `seg` `segs` `capacity`
Borrows and regions — chapter 12 · chapter 18	`ref` `mut_ref` `deref` `borrow` `region` `stack_new` `push` `pop` `swap`
Contracts and errors — chapter 14	`range` `ret` `expect` `panic`
Computation leaves — stand only after `call_builtin`	`clmul_lo` `clmul_hi` `aes_round` `aes_round_last` `aes_ctr` `ghash` `chacha20` `poly1305` `aes_gcm`

	chacha_poly sha256 sha384 sha512 crc32 hash_bytes rng_next
C strings — chapter 29	cstr_of str_from_cstr
Host leaves — chapter 32	file_open file_read file_write file_seek file_close file_type link_type dir_open dir_read dir_close dir_make path_remove path_rename net_listen net_accept net_connect net_resolve net_send net_recv net_close net_pair net_port env_get reactor_new r_read r_write

Table 50.3 — Kinds of builtins

★ **The fifteen computation leaves are not global words.** They stand only in that position — `call_builtin sha256 msg out`. Calling one bare is `E-BUILTIN-BARE`; naming something else after `call_builtin` is `E-BUILTIN-NAME`. The reason is one: a word a program uses once should not cost every reader a name to remember. The same rule already governs the stage names inside `pipe` and the type slot of `cast u8 x`.

This table sorts every builtin of the canon into a group. The chapter next to each group name explains that group with examples. The one-line meanings are authoritative in the repository's `docs/spec/BUILTIN-MEANINGS.tsv`.

Effects and capabilities

Effect	Permitting capability	Chapter
io	cap io · file_system · net · tty · clock · random	chapter 16
alloc · heap	cap allocator · cap heap	chapters 18 and 20
atomic	cap atomic	chapter 27
device	cap mmio	chapter 30
unsafe	cap c · cap machine (depending on use)	chapters 29 and 30
state · panic · wait · concurrent	Written without capabilities	chapters 15 and 26

Table 50.4 — Effect atoms and their paired capabilities

Appendix B — Diagnostics index

The first two tables list the diagnostics this book's examples actually trigger. Diagnostics you can meet but the examples do not show are gathered at the end under "Other diagnostics". A code keeps its meaning across releases; the compiler explains the details and the repair at the site (`--diag-json` in a machine-readable form).

At translation time

Code	Meaning	Shown in

E-ACCESS-MODE	the body breaks the read or write promise made with `access`	chapter 27
E-ACTOR-FIELD	an actor’s state field is read from outside	chapter 25
E-ACTOR-STATE-REF	a borrow is kept in an actor state field	chapter 25
E-ACTOR-UNINIT	a just-spawned actor’s slice state field is read before it is set	chapter 20
E-ALLOC-AMBIGUOUS	more than one fitting allocator and none chosen with `using`	chapter 20
E-ALLOC-NESTED	allocated from an outer source while an inner region of the same root is open	chapter 18
E-ALLOC-NOCAP	declares `alloc` but receives no allocation capability	chapter 16
E-ALLOC-NOSOURCE	a call draws from an allocator but no fitting one is visible in this op	chapter 20
E-ALLOC-OUTLIVES	region bytes handed to an actor born outside the region	chapter 18
E-ALLOC-SHARED	an allocator whose cursor is not atomic is handed to a task	chapter 26
E-ALLOC-TASK	an op that carves from a root is spawned as a task	chapter 26
E-ALLOC-USING-UNUSED	`using` on a call that does not draw from an allocator	chapter 20, chapter 34
E-ASM-TARGET-UNKNOWN	the machine name in an `asm` clause is not a known target	chapter 30
E-ASM-UNBOUND	the assembly template names an undeclared operand	chapter 30
E-ATOMIC-NOCAP	declares `atomic` but receives no `cap atomic`	chapter 27
E-ATOMIC-ORDER	a memory ordering that has no meaning for that atomic operation	chapter 27
E-BLOCK-UNCLOSED	a block opened with `do` is never closed with `end`	chapter 2
E-BOUND-UNSAT	the type argument does not satisfy the required trait	chapter 22, chapter 23
E-BRAND-REUSED	a brand opens a second storage	chapter 35
E-CAP-FORGE	spawns an actor holding a capability without holding that capability	chapter 16, chapter 20, chapter 25
E-CAP-KIND	a capability of the wrong kind was handed over	chapter 16

E-CAP-LOCAL	a received capability was copied into a local name	chapter 16
E-CAP-MISSING	a capability-requiring builtin was not given its capability as the first operand	chapter 16
E-CHAR	a character (symbol) this language does not have	chapter 6, chapter 7, chapter 8, chapter 9
E-CHAR-WIDTH	a character literal is not a single character	chapter 3
E-CLAUSE-ORDER	the clauses of an op header are out of the fixed order	chapter 2, chapter 3, chapter 22
E-COMPTIME-ARG	a run-time value in a `comptime` position	chapter 22
E-CONC-ALONE	a task group spawns one task that waits for a peer	chapter 26
E-CONC-DEADLOCK	every task in the group receives and none sends	chapter 26
E-CONFIG-TYPE	the configuration gives a value the option does not offer	chapter 31
E-CONFIG-UNDEF	a configuration or `config` names an undeclared build option	chapter 31
E-CONTRACT-DEAD	declares an error that `requires` already excludes	chapter 14
E-CONTRACT-IMPOSSIBLE	a call with constant arguments breaks the callee's `requires`	chapter 14, chapter 35
E-CONTRACT-UNSAT	two preconditions that cannot both hold	chapter 14
E-DEP-MISSING	the file at a `use … from` place cannot be read	chapter 21
E-EFFECT	performs an effect its `effects` clause does not declare	chapter 15, chapter 20, chapter 26
E-EFFECT-CALC	a pure `fn` performs an effect	chapter 1, chapter 5, chapter 15, chapter 16, chapter 18, chapter 24
E-EFFECT-DUP	the same atom twice in an `effects` clause	chapter 15, chapter 44
E-EFFECT-NO-CAP	declares `io` (or similar) without a capability that authorises it	chapter 2, chapter 16
E-EFFECT-NONE-MIX	`none` written together with a real effect	chapter 15
E-EFFECT-PURITY	a `fn` writes state visible to its caller or actor	chapter 5, chapter 15, chapter 25
E-EFFECT-REDUNDANT	`effects none` written on a `fn`	chapter 5

E-EFFECT-UNDEF	an effect word outside the closed vocabulary	chapter 15, chapter 44
E-ENS-UNDEF	`ensures` names something undefined — the returned value is `ret`	chapter 14
E-ENTRY-PARAMS	the entry point takes a non-capability input	chapter 16
E-ENUM-DOT	an enum variant not closed with .	chapter 10
E-ENUM-INFINITE	a variant embeds its own type by value	chapter 10
E-ERR-UNDECLARED	returns an error not in its `errors` clause	chapter 11
E-ERRORS-STATE	an `errors` condition reads an actor state field	chapter 25
E-ESCAPE	a reference to a local escapes the op	chapter 12
E-EXCL	overlapping borrows or owner access to the same value	chapter 12, chapter 20, chapter 26, chapter 43
E-EXPR-APP	a call inside an `expr` island is not parenthesised	chapter 8
E-EXPR-CHAIN	comparisons chained in an `expr` island	chapter 3
E-EXPR-UNARY	a unary operator in an `expr` island	chapter 8
E-FFI-LINK	an `extern` op does not name the C symbol it calls	chapter 29
E-FFI-NOCAP	calls C without receiving `cap c`	chapter 29
E-FFI-NOEFFECT	an op calling C declares no effect	chapter 29
E-FFI-NOUNSAFE	calls C without the `unsafe` mark	chapter 29
E-FFI-TYPE	a type the C ABI cannot express crosses the boundary	chapter 29
E-FIELD-GLUED	tried to read a field by gluing a dot to a value — write `field`	chapter 3
E-FN-CAP	an op to be passed as a callback requires a capability	chapter 29
E-FN-NOTEXPORT	`unsafe_fn` names an op that is not `export extern`	chapter 29
E-FOLD-OP	a `pipe` stage names an op that does not exist	chapter 24
E-FOLD-ORDER	a fold stage op declares its accumulator with a type other than its result	chapter 24
E-GROUP-UNCLOSED	an unclosed parenthesis	chapter 3

E-GUARD-FALLTHROUGH	the `else` of a `guard` does not leave	chapter 7
E-HEAP-NOHOST	asks for the growing root on a free-standing target	chapter 18
E-IF-VALUE	`if` used as a value	chapter 6
E-IMMUTABLE	`set` on a name bound with `let`	chapter 6
E-IR-ARITY	an op is called with the wrong number of arguments	chapter 5, chapter 48
E-IR-UNDEF	a name unknown at that place	chapter 6, chapter 7, chapter 21, chapter 22, chapter 25
E-ISR-CALLED	an interrupt handler called from code	chapter 30
E-ISR-EFFECT	an interrupt handler does not declare `effects device`	chapter 30
E-ISR-PARAMS	an interrupt handler takes parameters	chapter 30
E-LET-NOVALUE	nothing after `be`	chapter 6
E-LOCK-NOTYET	shared lock types are not built yet	chapter 26
E-MATCH-INEXHAUSTIVE	a `match` does not cover every case	chapter 7, chapter 10
E-MATCH-REDUNDANT	a `match` arm can never run (an arm after _, overlapping ranges)	chapter 11
E-METHOD-RECV	an attached op's receiver is not its first input	chapter 23
E-METHOD-UNDEF	no op of that name is attached to the receiver's type	chapter 22, chapter 23
E-MMIO-BYVALUE	an mmio register block taken by value	chapter 30
E-MMIO-PERM	writes a read-only register	chapter 30
E-MONO-NOTYPE	a generic call gives no leading type argument	chapter 22
E-MREF-SLICE	`mut ref slice` used	chapter 12
E-NAME-BUILTIN	a builtin op's name is used for a declaration or local	chapter 5, chapter 6
E-NAME-DUP	the same name declared twice in one module	chapter 21
E-NAME-SCOPE	a name declared inside a block is read outside it	chapter 6
E-NAME-SHADOW	re-binds a live name (shadowing)	chapter 3, chapter 6
E-OPT-UNUSED	a declared build option that no code reads	chapter 31
E-OWN-INCOMPLETE	a value that needs completion is dropped implicitly	chapter 19, chapter 28, chapter 36

E-OWN-JOIN	ownership state differs between branches	chapter 19
E-OWN-MOVED	uses a value after it was moved	chapter 19, chapter 25, chapter 28, chapter 35, chapter 36
E-PAR-ASSOC	the reduction operator is not associative	chapter 27
E-PAR-CARRY	a split loop writes a local that lives across iterations	chapter 27
E-PAR-FLOAT	a floating-point accumulation declared splittable	chapter 27
E-PAR-IDENTITY	a `reduce` starts from a value that is not the operator's identity	chapter 27
E-PAR-NOLOOP	the `parallel` clause finds no loop to split	chapter 27
E-PAR-READ	a split loop reads another iteration's element	chapter 27
E-PAR-WRITE	a split loop writes another iteration's element	chapter 27, chapter 45
E-PIPE-NO-TERMINAL	a stage after the terminal	chapter 24
E-PIPE-STAGE	a word that is not a `pipe` stage	chapter 24
E-PROFILE-LEVEL	uses concurrency the build profile does not provide	chapter 25
E-REGION-ESCAPE	carries region bytes out of the region	chapter 18
E-RETURN-PARTIAL	some path does not return a value	chapter 3, chapter 5, chapter 7
E-SPAWN-SCOPE	spawns a task outside a `task_group`	chapter 26
E-STMT-ELSE	`else` sits inside the block	chapter 7
E-TIER-EFFECT	an effect the declared machine tier cannot carry	chapter 30
E-TOPLEVEL	something that cannot appear at top level is at top level	chapter 3
E-TRAIT-EFFECT	the implementing op has more effects than the signature	chapter 23
E-TRAIT-MISSING	an op required by the trait is missing	chapter 23
E-TRAIT-SIG	`fn`/`proc` in a signature, or a parameter-count mismatch	chapter 23
E-TRY-NORESULT	`try` used in an op that cannot return that error	chapter 11
E-TYPE-ARG	an argument's type does not match the parameter	chapter 12

E-TYPE-ARGMUT	a read-only value is passed where it would be written (`mut_ref` of a `let`)	chapter 9, chapter 12
E-TYPE-ARRAY	`array` length written after the type	chapter 9
E-TYPE-BITCAST	`bit_cast` target is not a plain scalar	chapter 20
E-TYPE-COLLECT	`collect into` would put a wider value into a narrower buffer	chapter 24
E-TYPE-COND	a non-boolean in a condition	chapter 4
E-TYPE-DECL	`be` in a type declaration	chapter 13
E-TYPE-FIELD	a field is missing or unknown when building a struct	chapter 10
E-TYPE-INSTANCE	mixes different instances of the same generic	chapter 35
E-TYPE-KIND	a type of the wrong kind for that operation (e.g. `cast` of `bool`)	chapter 8, chapter 10, chapter 13
E-TYPE-MIX	float and integer literals mixed in one calculation	chapter 4
E-TYPE-MUT	writes an element of a non-`mut` slice	chapter 9
E-TYPE-NOMINAL	mixes nominally distinct types	chapter 13
E-TYPE-REF	writes through a shared `ref`	chapter 12
E-TYPE-REFVAL	a reference is used as if it were a value	chapter 12
E-TYPE-RETURN	the returned value does not match the op's output type	chapter 11, chapter 31
E-TYPE-SIGN	mixes signs with no value-preserving widening	chapter 4, chapter 39, chapter 40
E-TYPE-WIDTH	the value does not fit the declared type	chapter 3, chapter 4, chapter 13
E-UNSAFE-UNDECLARED	declares the `unsafe` effect but the op is not marked `unsafe`	chapter 29
E-VEC-SPLAT	`splat` was written inline in an expression	chapter 27
E-VISIBILITY	reaches a non-exported name of another module	chapter 21
E-VOCAB-REMOVED	a removed word or spelling	chapter 3, chapter 5, chapter 7, chapter 15, chapter 48
E-WIDEN-SIGN	`widen` from a signed to an unsigned type	chapter 13
W-EFFECT-OVER	an effect declared but never performed (warning)	chapter 15, chapter 16

W-EXPORT-HIDDEN	an exported signature names a type the module keeps to itself	chapter 21
W-NOT-YET	a word accepted by name that has no meaning yet	chapter 13
W-USE-EXTERNAL	imports a module not in the compilation unit (warning)	chapter 32

Table 50.1 — Diagnostics shown by the examples

When execution stops

Code	Meaning	Shown in
E-VM-OVERFLOW	integer overflow at the declared width	chapter 4
E-VM-DIV0	division by zero	chapter 4
E-VM-CAST	a narrowing or `cast` value does not fit	chapter 4
E-VM-SHIFT	shift amount not smaller than the width	chapter 4
E-VM-BOUNDS	index out of bounds	chapter 9
E-VM-NONE	took the value out of `none`	chapter 11
E-VM-CONTRACT	a contract was broken	chapter 14
E-VM-PANIC	the program called `panic`	chapter 7
E-VM-ANALYSIS	an eliminated check was actually needed (compiler bug)	chapter 40
E-TEST-FAIL	an `expect` in a test is false	chapter 31

Table 50.2 — Run-time diagnostics

Other diagnostics

These do not appear in the examples but can come up. Most are stops by the **VM** (the executor) when running with `--run`. The VM exists for tests and cross-checking, so it holds its resources at fixed sizes, and past a limit it **stops rather than quietly giving a wrong value**. A limit is the VM's size, not a defect of the program — the same program built natively usually runs fine.

code	meaning and repair
`E-IR-EXTRA`	operands are left over after an `expr` island — an island ends with one expression. Group with parentheses or split the line
`E-IR-LIMIT`	a compilation unit has more ops than the tool can carry — split the module
`E-IR-LOCALS`	one op has too many local names — split the op
`E-MONO-FIXPOINT`	generic monomorphisation does not finish (the type keeps growing) — break a generic that calls itself at a bigger type
`W-COL0`	a top-level declaration does not start at column 0 — the line may have ended up in the previous declaration's body
`W-UNBOUND`	a named `contract` is declared but no op names it with `satisfies`

W-RFC-PENDING	a feature whose design is still at the RFC stage was used — nothing in the program to fix
W-CBE-SLOW	this op is lowered to the slow path (tagged values) in native code — `--why-slow` says why, op by op

Table 50.3 — At translation time — not in the examples

code	meaning and repair
E-VM-ERR	`ok_value` was taken from a `result` that is an error — ask `is_ok` first or pass it on with `try`
E-VM-DANGLING	a borrowed reference was used after the place it borrowed from ended (use after return)
E-VM-EXCL	a writing borrow of a place overlapped another borrow — end the first borrow before taking the next
E-VM-READONLY	wrote through a read-only reference (`ref`) — borrow with `mut_ref` to write
E-VM-FIELD	read a field that the record or register block does not have
E-VM-AWAIT	`await` was given something that is not a job handle — handles come from `spawn <op>`
E-VM-CHAN-EMPTY · E-VM-CHAN-FULL	received from an empty channel or sent to a full one with no scheduler to wait — a wait that can never end
E-VM-SCHED-LOOP	handlers keep sending each other messages, so the scheduler cannot settle
E-VM-FD · E-VM-FMODE · E-VM-FWHENCE	a file number that cannot be written (only 1 · 2) · an unknown open mode (0 · 1 · 2) · an unknown `seek` origin (0 · 1 · 2)
E-VM-FHANDLE · E-VM-SHANDLE · E-VM-REACTOR	the value passed is not an open file / socket / reactor handle
E-VM-EXTERN	an `extern` op whose body is in C was called in the VM — build natively to run it
E-VM-MMIO · E-VM-RESERVE	a register block at an absolute address / storage handed over by the linker was opened in the VM — they mean something only on the real machine

Table 50.4 — At run time — a mistake in the program

code	what filled up
E-VM-STACK · E-VM-DEPTH · E-VM-OOM	the operand stack · the call depth · memory to grow the call stack — recursion is too deep
E-VM-BOXPOOL · E-VM-RECPOOL · E-VM-STKPOOL · E-VM-BSETPOOL	the pools for values wrapped by `some`·`ok` · struct values · stack containers · bitsets — keep fewer values alive at once

`E-VM-CHAN` · `E-VM-MBOX`	64 channels · 4096 async mailbox slots — they are not reused, so one run makes at most that many
`E-VM-FILEPOOL` · `E-VM-SOCKPOOL`	how many files · sockets can be open at once — close the ones you are done with
`E-VM-CHAIN-LIMIT`	VM activations nested inside VM activations went too deep
`E-VM-BUDGET`	the step budget of a cross-checking (oracle) run ran out — it does not happen in a normal `--run`

Table 50.5 — At run time — a VM limit (not a defect of the program)

Appendix C — Common mistakes and how to fix them

These are the places people often trip when writing Lowent with habits from other languages. The meanings of diagnostic codes are in Appendix B, and explanations are in the chapters named in the tables.

The surface

Mistake	Diagnostic	Fix
Writing `output` before `input`	`E-CLAUSE-ORDER`	Follow the clause order table. Clauses other than inputs are moved by `--fmt` (chapter 3)
Opening a block with a period or line break, as in `struct p .`	`E-STMT-NODO`	`struct p do … end` (`--fmt` fixes it)
Closing with `;` or `,`	`E-VOCAB-REMOVED`	The detached period `.`
Reading a field with `p.x`	`E-FIELD-GLUED`	`field p x`
`for x in xs`	`E-VOCAB-REMOVED`	`for x xs do … end`
`let x f64 be .5 .`	`E-LET-NOVALUE`	`0.5`
Naming a local the same as a parameter	`E-NAME-SHADOW`	Pick a new name
Using names like `count`, `text` or `len`	`E-NAME-BUILTIN` and others	Builtins and heredoc words cannot be names
A module name equal to an op name	`E-NAME-DUP`	Rename the module
`expr a lt b and b lt c`	`E-TYPE-LOGICAL`	`expr (a lt b) and (b lt c)` (chapter 8)
`type pct be u8 .`	`E-TYPE-DECL`	`type pct u8 .`

Table 50.1 — Mistakes on the surface

Values and flow

Mistake	Diagnostic	Fix
`set` on a `let`	`E-IMMUTABLE`	Make it with `var`
Using an integer as a condition	`E-TYPE-COND`	Write the question, as in `gt n 0`
Expecting overflow to wrap as in C	`E-VM-OVERFLOW` (at run time)	Choose the outcome with `wrap_add`, `sat_add` or `chk_add`
Expecting `narrow u8 x` to truncate	`E-VM-CAST` (at run time)	`narrow_wrap`, `narrow_sat`, `narrow_try`
`guard`'s `else` does not leave	`E-GUARD-FALLTHROUGH`	`return`, `break`, `continue`, `panic`, or use `if`
No `return` on some path	`E-RETURN-PARTIAL`	Return on every path
Using `if` as a value	`E-IF-VALUE`	Make a `var` and `set` it in each branch
Missing cases in `match`	`E-MATCH-INEXHAUSTIVE`	The missing `case` or `case _ .`
Writing enum variants one per line without periods	`E-ENUM-DOT`	`red .` for each variant
Extracting with `some_value` without checking	`E-VM-NONE` (at run time)	`guard is_some · value_or · match`
`array u8 4`	`E-TYPE-ARRAY`	`array 4 u8`

Table 50.2 — Mistakes with values and flow

Effects, capabilities and memory

Mistake	Diagnostic	Fix
Printing from a `fn`	`E-EFFECT-CALC`	`proc` with `effects io`
Writing `effects none` on a `fn`	`E-EFFECT-REDUNDANT`	Delete the clause
A `fn` writing to the caller through a `mut` parameter	`E-EFFECT-PURITY`	Make it a `proc`
`effects io` without receiving a capability	`E-EFFECT-NO-CAP`	`input out cap io .`
Receiving a capability but not passing it to the builtin	`E-CAP-MISSING`	As the first operand, as in `write_out out 1 …`
The entry point receiving data	`E-ENTRY-PARAMS`	The entry point receives only capabilities. Arguments via `cap args`
Writing to a slice without `mut`	`E-TYPE-MUT`	`mut slice`
Writing through `ref`	`E-TYPE-REF`	`mut_ref`
Borrowing the same value for writing twice	`E-EXCL`	Split the borrows or split the order
Returning a reference to a local	`E-ESCAPE`	Return a value or store it in the caller's storage

`mut ref slice`	`E-MREF-SLICE`	Return a new slice
Carrying a region's bytes outside	`E-REGION-ESCAPE`	Widen the region or carry out only values
Not closing a file or buffer	`E-OWN-INCOMPLETE`	Call `close` or `finish`. To discard, `drop`
Using a moved value again	`E-OWN-MOVED`	Use it before handing over, or get it back
Using the heap on a target without an operating system	`E-HEAP-NOHOST`	A fixed window (`cap allocator`) and `alloc`

Table 50.3 — Mistakes with effects, capabilities and memory

Abstraction and concurrency

Mistake	Diagnostic	Fix
Writing `fn` or `proc` in a trait signature	`E-TRAIT-SIG`	`area input s self . output u64 .`
Satisfying a pure signature with a `proc` without an effects line	`E-TRAIT-EFFECT`	Use `fn`, or write `effects`
A type argument not satisfying the trait	`E-BOUND-UNSAT`	Add `satisfies` and the op
Giving a run-time value where `comptime` is required	`E-COMPTIME-ARG`	A literal or module constant
`spawn <op>` outside a `task_group`	`E-SPAWN-SCOPE`	Wrap it in `task_group do … end`
Writing an accumulator in a split loop	`E-PAR-CARRY`	`reduce <place> <op> .`
Using a stage like sorting in `pipe`	`E-PIPE-STAGE`	Use only the seven stages and five terminals
Calling an effectful attached op with `method` inside a `fn`	(not rejected in this edition)	Call it directly as `<type>.<name>` (chapter 23)

Table 50.4 — Mistakes with abstraction and concurrency

Appendix D — Grammar summary

The normative text is the specification; this appendix is a reference gathering frequently sought shapes. There is no complete formal grammar yet.

Clause order of an op head

```
[export] [unsafe] [extern] fn|proc <name>
  satisfies · lowdoc                      ① what it is
  vector · priority                       ② character of the whole op
  input <name> comptime …                 ③ translation-time inputs
  input <name> cap … · region …           ④ capability and region inputs
```

```
  using <name> <type> .                    ⑤ allocator drawn from
  input <name> <type> .                    ⑥ data inputs
  output <type> .                          ⑦
  effects <atoms…> [via <type>] .          ⑧
  link "<name>" · variadic · asm <machine> ⑨ joining the outside
  access · inplace <written> <read> · invalidates <input> · parallel <name> split · reduce  ⑩
how it touches memory · runs split
  requires [static|debug|assume] <cond> .  ⑪ input conditions
  ensures <cond> .                         ⑫ output promises (ret)
  errors <variant> [<cond>] .              ⑬ failure
  tests … · schedule …                     ⑭
do
  <statements>
end
```

An `extern` op whose body is in C holds the same clauses, in the same order, **inside** `do … end` — `unsafe extern proc <name> do <clause>* end`. Only clauses go in that block.

Declarations

```
module <name> .
use <module> [from "<place>"] .
type <name> <type> .
newtype <name> <type> .
struct <name> do [satisfies <trait> .] [layout packed .] [mmio <address> .] <field> <type> [big|
little] [rw|ro|wo] . … end
enum <name> do <variant> [<field> <type>]… . … end
trait <name> do <op name> <clauses…> . … end
contract <name> do requires <cond> . … end
actor <name> do [satisfies …] state do <field> <type> . … end [failure restart max <n> .]
[mailbox bounded <n> .] <op>… end
build profile <name> .        build tier t0|t1|t2|t3 .        build <mode> .
build option <name> bool|int|choice … default <value> .
test <name> [schedule explore_interleavings [limit <n>] .] do expect <cond> . … end
```

Statements

```
let <name> [<type>] [using <source>] be <expr> .
var <name> [<type>] be <expr> .
set <place> <expr> .                       (place = name · field … · index …)
if <cond> . do … end [else do … end] .
while <cond> . do … end
for <name> <slice> do … end
guard <cond> . else <leaving statement> .
match <value> do case <pattern> [when <cond>] . do … end … end
return [<expr>] .    break .    continue .    panic "<text>" .
drop <name> .
region <name> stack|frame|arena|static|heap|mmap|disk|device do … end
task_group [cancel_on_error] do … end
pipe <source> do <stage> . … <terminal> . end
spawn send <actor> <message> <value>… .    drain <actor> .    schedule .
```

Patterns

```
case _ .                    everything else
case <integer> .  case <low> to <high> .
case <variant> [<name>…] .  case <pattern> or <pattern> .
case some <name> .  case none .  case ok <pattern> .  case error .
case <name> when <cond> .
```

Expressions

```
<op> <args>…                            prefix --- no precedence, forms inside forms in
parentheses
expr <a> + <b> * <c>                    infix island --- * / above + -, comparisons below, and
above or
make <type> do <field> <expr> . … end   <enum>.<variant> <values>…
field <value> <steps>…   index <slice> <n>   method <value> <name> <args>…
some <value>   ok <value>   error <variant>   none
try <expr> [else_none | else_error <variant>]
comptime <expr>   size_of <type>   config <name>
spawn actor <type>   send <actor> <message> <values>…   spawn <op> <args>…   await <handle>
alloc_bytes <root> capacity <n>
```

Heads and closers

A form starts with one head and ends with one closer. The closer is the detached period `.`, and `)` also closes what is open inside it. `do … end` is a pair of braces, so `end` closes only its own `do` — a construct that owns a block ends at its `end`, and a statement that uses a block value ends with its own `.`. A newline is not a closer but whitespace. Parentheses cannot cross block boundaries.

Appendix E — Standard library modules

Part IX (chapters 32–37) toured the standard library by use. This appendix is a **reference**, one page per module, gathering what each does, which ops it offers and what it does not do. It is not a place to read from the start but a place to open when needed.

Module pages share one shape. The head gives the source file, layer and capabilities received, followed by what the module does, its ops at a glance, how to use it, design and boundaries, and what is not built. The maturity each module writes (`experimental`, `incubating`, `standard`) is authoritatively in the source head and the development repository's ledger and moves between editions, so it is not copied here. The conventions every module follows — the caller's buffer, failure as a value, all or nothing, ownership that must be settled — are in chapter 32.

Module names may differ from file names. What `use` looks for is the `module` declaration in the file (`lib/str.low` is `strings`, `lib/alloc.low` is `allocs`). Cross-module names are always called qualified (`strings.find`), and a frequently used one can get an alias with `use codec as c .`.

Module	In one line
`strings`	String view operations — cut, find, compare
`strbuf`	Owned string buffer and null-terminated `cstr`

`fmt`	Formatting assembled into the caller's buffer
`utf8`	UTF-8 code point iteration and validation
`utf16`	UTF-16 surrogate arithmetic
`unicode`	Unicode property tables — extracted mechanically
`codec`	Hex · base64
`regex`	Pike VM regular expressions without backtracking
`term`	The pure half of a terminal renderer — ANSI assembly · screen diff · width
`sortlib`	In-place quicksort for `u64`
`sortgen`	Generic sort — the type brings the comparison
`searchlib`	Binary search over sorted slices
`hashmap`	`u64 → u64` open addressing
`strmap`	Byte string → `u64` hash map
`vecs`	Growing byte vector
`spsc`	Lock-free single-producer single-consumer ring buffer
`hash`	Hash-map slots (FNV-1a) · damage detection (CRC-32) · SHA-256
`math`	Floating-point maths — `close` instead of `==`
`random`	Separates reproducible sequences from OS entropy by name
`hmac`	HMAC-SHA256 · HKDF — who sent it
`chacha`	ChaCha20 stream — not safe alone
`poly`	Poly1305 — a fresh key per message
`aead`	ChaCha20-Poly1305 seal and open — cryptography starts here
`x25519`	Key agreement
`aes`	AES-128 block cipher — usually wrong used alone
`gcm`	AES-128-GCM — never repeat a nonce
`bigint`	Big-number modular arithmetic
`rsa`	RSASSA-PSS verification
`p256`	NIST P-256 curve and ECDSA verification
`p384`	NIST P-384 curve and ECDSA verification — the public web's intermediates
`ecdsa`	ECDSA P-256 signing — nonce derived without randomness
`ed25519`	Ed25519 signature verification
`der`	Minimal DER parser — extracts public keys only
`pem`	Unwrapping PEM envelopes
`x509`	Reading X.509 certificates — offsets and lengths only
`verify`	Certificate signatures and one link of a chain
`tls13`	TLS 1.3 parts — key schedule · records · transcript · Finished
`tlssrv`	Server TLS handshake and application data records
`tlscli`	Client TLS 1.3 handshake — the caller checks certificates
`http`	HTTP/1.1 request parser — its heart is rejection
`soa`	A trial of laying structs out as per-field arrays

Table 50.1 — L0 — pure computation (`effects none` · the caller's buffer)

Module	In one line
`allocs`	Allocator trait · bump allocators · two default allocators
`pool`	Generational-handle block pool
`shard`	Tokens splitting storage into non-overlapping pieces
`budget`	Bit budgets for handles and retiring generations
`wire`	Fields in one word — a mask carries position and width
`flags`	Named on/off settings in one word
`segarena`	Fixed-size segment arena
`pagecache`	Page ids and pin tokens
`growvec`	Growing byte vector — short name for `vecgen.vec u8`
`vecgen`	Generic growing vector `vec t`
`mapgen`	Generic hash map `table k v`
`nodelist`	Fixed-size intrusive list
`segview`	Cursor, total length and flattening for scattered views
`lifemode`	`lifemode` · `lifeatom` — when a value ends

Table 50.2 — L1 — storage (needs the allocation capability or borrowed bytes)

Module	In one line
`io`	Stream reading over slices
`outbuf`	Buffered output — forgetting to flush is rejected
`files`	Files and directories — forgetting to close is rejected
`tty`	Terminal input — raw mode · reading keys · pure key parsing
`net`	Sockets — TCP loopback · in-process connection pairs
`clock`	Time and deadlines — monotonic and wall clocks
`trust`	Finding a trusted root in the trust store — the bundle is streamed

Table 50.3 — L2 — host (needs capabilities)

The cryptographic modules are stacked in layers towards TLS 1.3; that order and the top that does not exist yet are in chapter 36.

`strings` — string view operations

Source	`lib/str.low`
Layer	L0 — pure computation
Capabilities	none

Compares, searches, trims and splits strings **without copying**. Use it to split input on spaces, strip a prefix, or find a substring. The result is always a **view** over the original — a window pointing at someone else's bytes — and not a single byte of new memory is used.

```
use strings .

if strings.starts_with line "GET " . do
  let rest slice u8 be strings.remove_prefix line "GET " .
end
```

`rest` points directly at part of `line`. This module was built only from what the language already has (`len`, `index`, `subslice`, `eq`, `option`, `guard`, `while`, `actor`) and adds no builtin. Most string operations need no ownership — cutting, searching and comparing only produce views. Work that needs a writable buffer, such as appending, belongs to `strbuf`.

Design and boundaries

- **What it does** — byte-level comparison, search, prefixes and suffixes, trimming and splitting. Every `fn` is `effects none` and allocates nothing.
- **It does not interpret encodings.** `str` is bytes. UTF-8 validation and decoding belong to `utf8`.
- **It does not own or modify.** It never changes a byte of the original.
- **There is no `str_len` or `str_sub`.** `str` is an alias of `slice u8`, so the builtins `len` and `subslice` work as they are. One meaning does not get two names.
- **No sentinel values.** Conventions like "−1 means not found" are not used. Not found is `none` of an `option`.

`type str slice u8 .` is a **transparent alias**. It is not a `newtype`, so string literals, `slice u8` values and `subslice` results all fit a `str` position as they are. Only the `str_splitter` actor holds state — the original `src`, the separator byte `sep`, the cursor `pos` and the exhausted flag `fin`.

Ops at a glance

Every `fn` is `effects none`, and the actor's two `proc`s are `effects state`. Call them qualified by module name, as in `strings.find`.

op	Signature	When it cannot
`eq_str`	`(a str, b str) → bool`	`false` if different
`has_byte`	`(s str, b u8) → bool`	`false` if absent
`find`	`(hay str, needle str, from u64) → option u64`	`none` if not found or `from > len hay`
`has`	`(hay str, needle str) → bool`	—
`starts_with`	`(s str, prefix str) → bool`	—

ends_with	`(s str, suffix str) → bool`	—
remove_prefix	`(s str, prefix str) → str`	**The original as is** if no prefix
remove_suffix	`(s str, suffix str) → str`	The original as is if no suffix
trim_start	`(s str, cut str) → str`	An empty view if everything is cut
trim_end	`(s str, cut str) → str`	An empty view if everything is cut
split_next	`(src str, sep u8, pos u64) → option str`	`none` if `pos > len src`
str_splitter	actor: `init(s2 slice u8, b u8) → u64 · next → option slice u8`	`next` is `none` after exhaustion

Table 50.1 — Ops of `strings`

Ops in detail

The `slice u8` passed in is a view, and what comes back is another view over the same bytes. That is why it is not `mut`, is `effects none`, and is valid only while the original lives. The two arguments of comparison and search are always (the thing searched, the thing sought) — `has_byte s b` reads "is `b` in `s`".

`eq_str` — byte-wise equality. If lengths differ it is `false` before looking at bytes. The builtin `eq` is scalar-only, so slice comparison lives here. That is why the name could not be shortened to `eq`.

`has_byte` — is the single byte `b` in `s`. A byte, not a string. The cut-set test of `trim_start` and `trim_end` uses it.

`find` — searches `hay` for `needle` from `from` and gives the start index as `some`. Why `from` exists is the character of this module: **the cursor is a value the caller holds.** The next search calls again with the position after the match as `from`. The library hides state nowhere, so scanning the same string from several places at once causes no interference. `from > len hay` gives `none`, and an empty `needle` gives `some from`.

`has` — when you only ask "is it in there". Inside it is `is_some (find hay needle 0)`.

`starts_with` · `ends_with` — prefix and suffix tests. If the piece is longer than `s`, it is `false` without looking.

`remove_prefix` · `remove_suffix` — a view with the prefix or suffix removed. If the piece **is not there, the original is returned as is**. That is not failure, so the return is not an `option`. If you need to know whether it was removed, ask `starts_with` or `ends_with` first.

`trim_start` · `trim_end` — a view with certain bytes stripped from the front or back. `cut` is not an ordered string but **a set of bytes** — `" \t"` means "space or tab", not "space followed by tab". If everything is cut, a zero-length view comes out.

`split_next` — when splitting `src` on the separator byte `sep`, gives as `some` the piece from `pos` up to the next `sep`. **Advancing the cursor is the caller's job** — on receiving a piece, push `pos ← pos + len(piece) + 1` yourself (+1 for the separator byte). The last piece with no `sep` after it is also `some`, and `pos > len src` giving `none` is the end signal. `"a,b"` gives `"a"`, `"b"`; `"a,"` gives `"a"`, `""` (a trailing empty piece); `""` gives one `""`.

`str_splitter` — the pull version of `split_next`. The actor's state holds the cursor. After putting the original and separator into `init`, `next` takes no arguments, giving pieces as `option` until it is exhausted and gives `none`.

Using it

The module name is the `module` declaration in the file. The module of `lib/str.low` is `strings`.

```
use strings from "lib/str.low" .   rem write the path directly (relative to the using file)
use strings .                      rem resolved from the standard module location
use strings as s .                 rem alias --- then call s.eq_str
```

Finding and splitting:

```
module demo .

use strings .

fn t_find output u64 . do
  let r option u64 . be strings.find "hello world" "world" 0 .
  guard is_some r . else return 99 .
  return some_value r .
end

fn t_split output u64 . do
  rem "aa,b,,cc" split on ',' (byte 44): "aa", "b", "", "cc" --- four pieces
  var pos u64 be 0 .
  var pieces u64 be 0 .
  var r option slice u8 . be strings.split_next "aa,b,,cc" 44 pos .
  while is_some r . do
    set pieces (add pieces 1) .
    set pos (add pos (add (len (some_value r)) 1)) .
    set r (strings.split_next "aa,b,,cc" 44 pos) .
  end
  return pieces .
end
```

`t_find` gives 6 and `t_split` gives 4. If you want state, use the actor version — the actor holds the cursor.

```
proc t_splitter output u64 . effects state . do
  var sp strings.str_splitter . be spawn actor strings.str_splitter .
  let d u64 be send sp init "one,two,three" 44 .
  var pieces u64 be 0 .
  var r option slice u8 . be send sp next .
  while is_some r . do
    set pieces (add pieces 1) .
    set r (send sp next) .
  end
  return pieces .
end
```

The only difference between the two is who holds the cursor. If the same original must be scanned along several paths at once, `split_next` fits; if a single loop just pulls to the end, the actor version is shorter.

Counter-examples

Counter-example. Comparing slices with the builtin `eq`

```
guard eq "abc" "abc" . else return 0 .        rem ✗ eq is scalar-only
```

It is `E-VM-TYPE`. Compare slices with `strings.eq_str`.

Counter-example. Not advancing the cursor in a split loop

```
var r option slice u8 . be strings.split_next src 44 pos .
while is_some r . do
  set r (strings.split_next src 44 pos) .   rem ✗ pos stays the same
end
```

It is not a translation error — **the program never ends.** Giving the same `pos` yields the same piece forever. Advance by `pos + len(piece) + 1` each time.

Counter-example. Using an `option` as a value

```
let r option u64 . be strings.find "abc" "zz" 0 .
return some_value r .                         rem ✗ no none check
```

It translates, and stops at run time with `E-VM-NONE` the moment nothing is found. Testing only with inputs that are found never reveals it. `guard is_some r . else …` comes first (chapter 11).

Counter-example. Changing the original a view points at, later

```
let piece slice u8 be strings.remove_prefix line "GET " .
set (index line 4) 88 .                                  rem ✗ the original was changed
```

With no error, the content of `piece` silently changes — because it is a window, not a copy. To hold on to the content, copy it with `strbuf`.

Cautions

- **It is byte-wise, not character-wise.** Characters made of several bytes, like Korean or emoji, are not one unit. A `needle` is found as a byte string as it is, but `cut` is a byte set and cannot treat a multibyte character as one unit. For characters, go to `utf8`.
- **A view shares lifetime and content with its original.** If the original changes, so does the view's content; if the original goes away, the view cannot be used. Read it as "received a window", not "received a value".
- **`remove_prefix` does not report failure.** If you need to know whether it removed anything, ask `starts_with` first.
- **`find` is a naive search.** Its cost is O(`len hay` × `len needle`). Account for it where very long strings are scanned repeatedly.
- **Cursor version and actor version.** `split_next` shares no state, so several cursors may read at once. `str_splitter` holds state, so whoever spawned it is responsible until it is exhausted.
- **There is no direct path to a null-terminated string.** A view cannot promise a trailing 0 byte. Go through `as_cstr` of `strbuf`.

- **Names clashing with builtins.** `eq` and `contains` are builtins, so this module named its ops `eq_str` and `has_byte`. Avoid builtin names for your own variables too.

`strbuf` — owned string buffer and null-terminated `cstr`

Source	`lib/strbuf.low`
Layer	L0 — pure computation (the caller's buffer)
Capabilities	none · only `as_cstr` is `unsafe`

Builds strings **by appending little by little**. Use it to assemble a path from a directory, "/" and a name, or to build a message from several pieces. If `strings` is views (reading), `strbuf` is writing — it appends, puts in as much as fits, and grows.

```
use strbuf .

var b strbuf.str_buf be strbuf.new .
let r result void strbuf.sb_error . be strbuf.append b buf "hello" .
guard is_ok r . else return 1 .
```

In this module a "buffer" is two pieces — the **state** `b` remembering how much was written, and the **place** `buf` where the bytes actually go. They are separate, so ops always receive both. Where the buffer comes from (`cap allocator`, a region, a static buffer) is the caller's business, and the library **never allocates behind your back**. The only builtin added is `cstr_of`, which gives the raw pointer of a slice.

Design and boundaries

What to do when room runs out is chosen by the caller, by name. It is the same discipline as `wrap_add`, `sat_add` and `chk_add`. No single op decides for itself depending on the situation.

op	Kind	Behaviour
append	All or nothing	If it does not fit, it refuses **without writing a single byte**
append_trunc	As much as fits	Gives the count written — being truncated comes out **as a value**
append_grow	Grow	**The caller provides** the larger place

Table 50.1 — Three kinds of appending

- **It does not own bytes.** `str_buf` holds only state (`len`), and the byte buffer is an argument on every call. Capacity written in two places would diverge, so it is not in the struct — the `len` of the `buf` the caller gives is the capacity.
- **It does not grow by itself.** Growth is expressed only by the caller giving a larger buffer to `append_grow`.
- **There is no direct path from a view to `cstr`.** A view cannot promise a trailing 0 byte. The only safe way to make a `cstr` is through `str_buf`, and that route makes allocation visible.

> **The null seal**
>
> Every append **reserves 1 byte** — the content capacity is `len(buf) - 1`. After writing it seals with `buf[len] = 0`, so `as_cstr` is O(1) and allocation-free. The buffer must be at least 1 byte; in practice, remember **a buffer of n + 1 bytes for n bytes of content**.

There are three types. `str_buf` (state `len u64`), `sb_error` (one failure name, `no_room` — failure is a value, not a stop), and `cstr` (a `newtype` of a raw pointer to a null-terminated `char*`, only for the boundary with other languages).

Ops at a glance

op	Signature	When it cannot
new	`fn () → str_buf`	—
as_str	`fn (b str_buf, buf slice u8) → slice u8`	— (a view, not a copy)
room	`fn (b str_buf, buf slice u8) → u64`	stops on underflow for an empty (length 0) buffer
as_cstr	`unsafe proc (b str_buf, buf mut slice u8) → cstr`	stops out of bounds if `len + 1 > len(buf)`
append	`proc (b mut str_buf, buf mut slice u8, s slice u8) → result void sb_error`	`error no_room` (buffer unchanged)
append_trunc	`proc (b mut str_buf, buf mut slice u8, s slice u8) → u64`	— (known from count written < `len s`)
append_grow	`proc (b mut str_buf, old mut slice u8, new_buf mut slice u8, s slice u8) → result void sb_error`	`error no_room` (original unchanged)

Table 50.2 — Ops of `strbuf`

Ops in detail

- **`new`** — an empty state with `len 0`. No parameters — only state is made.
- **`as_str`** — a view of the bytes written so far, i.e. `subslice buf 0 len`. The null is not in the view — the length is the truth.
- **`room`** — remaining content capacity `len(buf) - 1 - len`. The 1 byte for the null is excluded. Use it to ask “can this much more go in” before appending.
- **`as_cstr`** — seals a null at position `len` and gives the start pointer as `cstr`. `buf` is `mut` because it **actually writes** the null byte. `cstr_of` is `effects unsafe`, so this op is `unsafe` too and cannot be called from pure code.
- **`append`** — if `len + len(s) + 1 ≤ len(buf)`, writes everything, seals and gives `ok`. Otherwise `error no_room`, and **not a single byte is written** (length and buffer unchanged). A half-written buffer is silently wrong content, which is worse than refusal.
- **`append_trunc`** — writes only as much as fits and returns the count written. Comparing it with `len s` tells whether it was truncated. Without comparing, you never know.
- **`append_grow`** — moving the old content to `new_buf`, appending `s` and sealing is one action. The `new_buf` parameter is the embodiment of “never allocates behind your back”. If even the new place is too small, `error no_room`, and **nothing is moved**. To grow you must be the one holding larger bytes.

Using it

The three kinds behave differently. The caller provides the buffer.

```
module demo .

use strbuf .

proc build input buf mut slice u8 . . output u64 . do
  var b strbuf.str_buf be strbuf.new .
  let r1 result void strbuf.sb_error . be strbuf.append b buf "ab" .
  guard is_ok r1 . else return 90 .
  guard eq (field b len) 2 . else return 91 .
  rem the seal --- a null at position len (buf[2])
  guard eq (index buf 2) 0 . else return 92 .
  rem buffer 5 = content 4 + null 1 --- 2 + 3 + 1 > 5, so refused and the buffer stays clean
  let r2 result void strbuf.sb_error . be strbuf.append b (subslice buf 0 5) "xyz" .
  guard is_error r2 . else return 93 .
  guard eq (field b len) 2 . else return 94 .
  rem in the same situation trunc writes as much as fits (2) and returns 2
  let n u64 be strbuf.append_trunc b (subslice buf 0 5) "xyz" .
  guard eq n 2 . else return 95 .
  return field b len .
end
```

Growth is expressed by the caller giving a new place.

```
proc grow input small mut slice u8 . . input big mut slice u8 . . output u64 . effects none . do
  var b strbuf.str_buf be strbuf.new .
  let r1 result void strbuf.sb_error . be strbuf.append b small "ab" .
  guard is_ok r1 . else return 90 .
  let r2 result void strbuf.sb_error . be strbuf.append b small "xyz" .
  guard is_error r2 . else return 91 .
  rem I provide the larger place --- the library does not allocate
  let r3 result void strbuf.sb_error . be strbuf.append_grow b small big "xyz" .
  guard is_ok r3 . else return 92 .
  guard eq (index big 0) 97 . else return 93 .
  guard eq (index big 2) 120 . else return 94 .
  return field b len .
end
```

`build` gives 4 (“ab” 2 + 2 written by trunc) and `grow` gives 5 (“abxyz”). The real use is path assembly — append with `append b pathbuf dpath`, `append b pathbuf "/"`, `append b pathbuf name`, then take the view with `as_str b pathbuf` and pass it to `files`.

Counter-examples

Counter-example. Calling `as_cstr` from pure code

```
fn f input b strbuf.str_buf . input buf mut slice u8 . . output u64 . do
  let p strbuf.cstr be strbuf.as_cstr b buf .   rem ✗ calling an unsafe proc from a fn
  return 0 .
end
```

Rejected at translation with `E-EFFECT-CALC`. Making a `cstr` is a guarantee of null termination, and that guarantee stands only inside an `unsafe` boundary. Make the caller an `unsafe proc`.

Counter-example. A buffer size forgetting the null

If the buffer is exactly 2 bytes, not even `"ab"` fits (2 + 0 + null 1 > 2). The very first `append` is `error no_room` and `field b len` stays at 0. Content of n bytes needs a buffer of n + 1 bytes.

Counter-example. Reading the buffer without checking the `result`

```
let r result void strbuf.sb_error . be strbuf.append b buf "hello" .
let v slice u8 be strbuf.as_str b buf .   rem ✗ r was not checked
```

No stop and no error. The buffer is not broken either — refusal does not touch it. You just get **a short string missing what you expected**. `guard is_ok r . else …` comes first.

Counter-example. Mixing state and buffer

```
let r result void strbuf.sb_error . be strbuf.append b buf1 "ab" .
let v slice u8 be strbuf.as_str b buf2 .   rem ✗ length from buf1, bytes from buf2
```

With no error, a view of the wrong bytes comes out. Pair one `str_buf` with one buffer only (or its successor moved by `append_grow`).

Cautions

- **Make the buffer 1 byte larger than the content.** Because of the null seal.
- **Refusal by `append` is quiet.** It is a value, not a stop, so without checking the `result` the program just keeps running.
- **After growing, discard the old buffer.** The moment `append_grow` gives `ok`, the truth is in `new_buf`. Do not pass `old` to later calls.
- **`as_cstr` is not a snapshot.** It gives the start pointer as is, so appending afterwards changes what it points to. Make it right before calling the other language and use it immediately.
- **`room` stops on a length-0 buffer** (`0 - 1`). The buffer is at least 1 byte.

`fmt` — formatting assembled into the caller's buffer

Source	`lib/fmt.low`
Layer	L0 — pure computation (the caller's buffer)
Capabilities	none

Prints numbers and text **into a buffer the caller provides** to make strings. Use it to assemble a line to output or one line of a log. This module makes no paper; it only writes on the paper handed to it (`mut slice u8`).

```
use fmt .

let p option u64 . be fmt.put_u64 buf 0 1234 .
guard is_some p . else return 1 .
```

`0` is `pos` (the next place to write), and the `option u64` returned is **the new `pos` on success, `none` if room ran out**.

Formatting is not output. Every op only assembles bytes into the caller's buffer and, without exception, is `effects none`. Actual output is done separately by a caller holding `cap io`, with `write_out` or `outbuf`. So assembly code can be called from anywhere without capabilities, tests compare buffers rather than screens, and the caller decides what to emit and when. Turning numbers into bytes needs neither kernel nor allocation, so it is a library — no builtin was added, and the whole file is inside the VM/native cross-check.

Design and boundaries

- **There is one convention — every writer is `(buf, pos) → new pos`.** Success gives `some newpos`, shortage gives `none`. It refuses with a value rather than stopping. Give the new `pos` to the next call as it is and the text joins up.
- **All or nothing.** If room runs out, not a byte is written — if "12" remains instead of "1234", nobody notices. That is why `put_u64` and `put_hex` count the digits first (`dec_width`, `hex_width`) and write from the front.
- **A line break is one LF** (`put_nl`). It is the same convention as `take_line` in `io`, and CRLF is neither made nor removed.
- **What it does not do** — floating-point formatting, padding and alignment, locale (thousands separators). If needed, measure the width with `dec_width` and fill spaces yourself with `put_byte`.
- **No state.** No struct, no enum. The buffer and cursor both belong to the caller, so any number of uses anywhere at once do not interfere.

Ops at a glance

op	Signature	When it cannot
`put_byte`	`proc (buf mut slice u8, pos u64, b u8) → option u64`	`none` if no room, buffer unchanged
`put_str`	`proc (buf, pos, s slice u8) → option u64`	`none` unless all fits, buffer unchanged
`dec_width`	`fn (n u64) → u64`	never fails

put_u64	`proc (buf, pos, n u64) → option u64`	`none` if no room, buffer unchanged
put_i64	`proc (buf, pos, n i64) → option u64`	`none` if no room
hex_width	`fn (n u64) → u64`	never fails
put_hex	`proc (buf, pos, n u64) → option u64`	`none` if no room, buffer unchanged
put_bool	`proc (buf, pos, b bool) → option u64`	`none` if no room
put_nl	`proc (buf, pos) → option u64`	`none` if no room

Table 50.1 — Ops of `fmt` — all `effects none`

`put_` in a name means “write into the buffer” and `_width` means “only count how many bytes are needed”. Those that change the buffer are `proc`, those that only count are `fn`, but none of the nine touches the outside.

Ops in detail

Every writer’s first two parameters are the same. `buf` is **where** to write (received because the module does not allocate, `mut` because it changes it), and `pos` is **from which place** in it (the module does not remember the cursor, so passing the new `pos` from the previous call is the only way to say “continue”).

- **`put_byte`** — writes the single byte `b` to `buf[pos]` (`'-'` is 45, LF is 10). Success gives `some (pos+1)`.
- **`put_str`** — writes the whole byte string `s`. `s` is only read, so a string literal can be given as is. `none` if `pos + len s` exceeds `len buf`.
- **`dec_width`・`hex_width`** — the number of digits when writing `n` in decimal or hex. At least 1 (`0` is one digit). They precompute for all-or-nothing, but you may use them directly to measure in advance “how many bytes are enough”.
- **`put_u64`・`put_hex`** — unsigned decimal and hex (lowercase, no `0x`). If you need `0x`, put `put_str buf pos "0x"` in front.
- **`put_i64`** — signed decimal. If negative, writes `-` first and handles the magnitude in `u64`. The i64 minimum (−9223372036854775808) cannot be negated to a positive, so `0 - n` is computed in `u64`. This boundary value prints correctly too.
- **`put_bool`** — writes `"true"` or `"false"`. Not `"1"`/`"0"`, so 4 or 5 bytes are needed.
- **`put_nl`** — one LF byte. Same as `put_byte buf pos 10`.

Using it

Assemble purely, output once at the end with `cap io`. The key is **passing the returned new `pos` on to the next call**.

```
module report .

use fmt .

proc main
  input out cap io .
  input al  cap allocator .
  output u8 .
  effects alloc io .
do
```

```
  let g option mut slice u8 . . be alloc_bytes al capacity 64 .
  guard is_some g . else return 70 .
  let buf mut slice u8 . be some_value g .

  rem assembly --- not a byte has gone out yet
  let p1 option u64 . be fmt.put_str buf 0 "answer=" .
  guard is_some p1 . else return 71 .
  let p2 option u64 . be fmt.put_u64 buf (some_value p1) 42 .
  guard is_some p2 . else return 72 .
  let p3 option u64 . be fmt.put_str buf (some_value p2) " hex=" .
  guard is_some p3 . else return 73 .
  let p4 option u64 . be fmt.put_hex buf (some_value p3) 255 .
  guard is_some p4 . else return 74 .
  let p5 option u64 . be fmt.put_nl buf (some_value p4) .
  guard is_some p5 . else return 75 .

  rem output --- only the assembled front part (0 … p5) goes out
  let w u64 be write_out out 1 (subslice buf 0 (some_value p5)) .
  return 0 .
end
```

The output is `answer=42 hex=ff` and one line break. Giving each piece a different exit code tells at once where room ran out. Measuring the buffer size in advance works with the same tools.

```
fn need_for input n u64 . output u64 . do
  let w u64 be fmt.dec_width n .
  return add (add 7 w) 1 .
end
```

Counter-examples

Counter-example. Outputting inside an `effects none` op

```
proc bad input out cap io . output u64 . effects none . do
  return write_out out 1 "x" .
end
```

The declaration becomes a lie and is rejected with `E-EFFECT` (`E-EFFECT-CALC` for a `fn`). Even changed to `effects io`, it is `E-EFFECT-NO-CAP` without a `cap io` input.

Counter-example. Discarding the returned `pos` and reusing the old one

```
let p1 option u64 . be fmt.put_str buf 0 "answer=" .
let p2 option u64 . be fmt.put_u64 buf 0 42 .        rem ✗ pos should be some_value p1
```

It translates, and the result overlaps like `42swer=` instead of `answer=42`. If the front of the output is mangled, first check whether `pos` was passed on.

Counter-example. Carrying on ignoring `none`

If room runs out, that piece **was never written**. Carrying on without `guard is_some … else return …` sends the output with the middle piece missing. If output goes wrong only for long values, the buffer is short. All or nothing guarantees one op, not a whole sequence of calls. Conversely, taking out with `some_value` without `guard` stops with `E-VM-NONE` the moment room runs out — testing only with short values never reveals it.

> **Counter-example. Giving the whole buffer to `write_out`**
>
> Giving `buf` instead of `subslice buf 0 (some_value p5)` sends the unwritten back part too. Mysterious bytes stick to the end of the line. The last `pos` is **the length written**.

Cautions

- **Running short is a normal path.** It is an expected answer, not an error, so receive it with `guard`. If you need an upper bound, measure in advance — `u64` decimal is at most 20 digits, hex at most 16, and `i64` adds one for the sign.
- **`pos` is both an index and a length.** After assembly, `pos` equals the bytes written so far, so `subslice buf 0 pos` is the result.
- **To reuse the buffer, just reset `pos` to 0.** Old content need not be cleared — emit only up to the current `pos` and the old bytes are not seen.
- **Write CRLF yourself.** If a protocol needs it, write `put_str buf pos "\r\n"`.
- **Every op is `effects none`.** They can be called inside a `fn` or an actor handler. Only output needs `cap io` and `effects io`.
- **Prefixes, signs and padding are the caller's job.** Attach `0x`, `+`, thousands commas or filler spaces yourself with `put_str` and `put_byte`.

`utf8` — UTF-8 code point iteration and validation

Source	`lib/utf8.low`
Layer	L0 — pure computation
Capabilities	none

Walks and checks UTF-8 byte strings **by code point**. A code point is the number Unicode gives a character (`'한'` = U+D55C = 54620). In UTF-8 one code point takes 1 … 4 bytes — one number is not one byte. Use it to count characters in a string mixing Korean and emoji, or to see whether input is valid UTF-8.

```
use utf8 .

let c option u64 . be utf8.decode s 0 .
guard is_some c . else return 1 .
```

`0` is not “the first character” but **byte offset 0**. Every position in this module is a byte offset, and advancing by character is done by `next`.

Why validation is needed. UTF-8 is variable-length, so not every byte string is valid. A sequence may be cut short, a continuation byte may be wrong, or the same character may be written in more bytes than needed — an **overlong encoding**. Overlong encodings are especially dangerous: if the same code point can be written as two byte strings, a filter like “does this string contain `/`” lies. It is a classic path for bypassing security filters.

Why it is a library. `str` is bytes (chapter 9), and UTF-8 was not enforced as a type invariant. Encoding is a choice of a higher layer, and this module is that layer. Rust enforces UTF-8 on `str` to get character iteration for free, at the cost of validation on every construction path. Here validation is paid **only when wanted**, and in exchange byte slicing (`subslice`) cannot fail. No builtin was added.

Design and boundaries

- **Invalid UTF-8 is answered with a value.** `none` means “from here it is not UTF-8”. It neither stops nor inserts a replacement character (U+FFFD) — replacement is an irreversible loss that takes the choice away from the caller.
- **What it rejects** — overlong encodings, surrogates (U+D800 … DFFF), values above U+10FFFF, truncated sequences, and continuation bytes in lead position. Surrogates are parts UTF-16 uses to split large characters over two units, so they have no place inside UTF-8 (`utf16`).
- **No partial answers.** `count_chars` gives `none` on meeting an invalid byte.
- **Not built** — normalisation, locale comparison, grapheme clusters (`term`), case conversion, encoding (code point → bytes). Iteration and validation only.

Code point range	Bytes	Lead byte	Continuation bytes
U+0000 … U+007F	1	`0xxxxxxx`	none
U+0080 … U+07FF	2	`110xxxxx`	`10xxxxxx` × 1
U+0800 … U+FFFF	3	`1110xxxx`	`10xxxxxx` × 2
U+10000 … U+10FFFF	4	`11110xxx`	`10xxxxxx` × 3

Table 50.1 — UTF-8 byte layout

Three things follow from the table. The lead byte alone gives the length (`seq_len`). `10xxxxxx` in lead position is not the start of a character. A value smaller than the minimum code point each length can hold (at least 128 for 2 bytes, 2048 for 3, 65536 for 4) is overlong — `decode` filters with these thresholds.

Ops at a glance

op	Signature	When it cannot
`is_cont`	`(b u8) → bool`	never fails
`seq_len`	`(b u8) → u64`	`0` if not a lead byte
`decode`	`(s slice u8, at u64) → option u64`	`none` if invalid, truncated or out of range
`next`	`(s slice u8, at u64) → option u64`	`none` if invalid, truncated or at the end
`count_chars`	`(s slice u8) → option u64`	`none` if any byte is invalid
`is_valid`	`(s slice u8) → bool`	never fails

Table 50.2 — Ops of `utf8` — all `fn · effects none`

Ops in detail

This module holds no state, so where and from where to read is always given by the caller.

- **`is_cont`** — is it a continuation byte (`10xxxxxx`, i.e. `eq (bit_and b 192) 128`). A judgement needing no context, so it takes one byte.
- **`seq_len`** — the byte length (1 … 4) of a code point from its lead byte. Length information lives only in the high bits of the lead byte. **0 is not a length but an error signal** — adding it as is leaves the cursor still and the loop never ends.
- **`decode`** — reads one code point at byte offset `at` and gives its **value**. `none` if `at` is past the end, the lead is a continuation byte, the sequence is cut off by the slice end, a continuation byte is not `10xxxxxx`, the encoding is overlong, it is a surrogate, or it exceeds U+10FFFF. A character spans up to 4 bytes, so it takes the whole slice, and `len s` is the basis for judging truncation. `at` must be the **start** of a character — feeding only positions `next` gave is a safe habit. It does not return the length.
- **`next`** — the **start position** of the next code point. `none` if invalid — it does not skip and paper over. It looks only at lead, length and truncation; range checks are `decode`'s job. Using the returned value as the next `at` is one step of iteration.
- **`count_chars`** — walks everything and gives the number of code points. `none` on an invalid byte. "Everything" is the contract, so it takes no start position. Cost is O(bytes).
- **`is_valid`** — is all of it valid UTF-8. It embodies "validation is a separate op, not a type invariant", implemented as `is_some (count_chars s)`. If you also need the count, calling `count_chars` once is better.

Using it

Validate first, read values with `decode`, advance with `next`. `"한"` is 3 bytes, 1 character, value 54620 (U+D55C).

```
module cpdump .

use utf8 .
use fmt .

proc main
  input out cap io .
  input al  cap allocator .
  output u8 .
  effects alloc io .
do
  let s slice u8 be "a한😀" .
  guard utf8.is_valid s . else return 65 .

  let g option mut slice u8 . . be alloc_bytes al capacity 128 .
  guard is_some g . else return 70 .
  let buf mut slice u8 . be some_value g .
  var pos u64 be 0 .

  var i u64 be 0 .
  while lt i (len s) . do
    let c option u64 . be utf8.decode s i .
    guard is_some c . else return 66 .
    let a option u64 . be fmt.put_str buf pos "U+" .
    guard is_some a . else return 71 .
    let b option u64 . be fmt.put_hex buf (some_value a) (some_value c) .
    guard is_some b . else return 72 .
    let d option u64 . be fmt.put_nl buf (some_value b) .
    guard is_some d . else return 73 .
    set pos (some_value d) .
    let nx option u64 . be utf8.next s i .
    guard is_some nx . else return 67 .
    set i (some_value nx) .
  end

  let w u64 be write_out out 1 (subslice buf 0 pos) .
  return 0 .
end
```

The output is three lines: `U+61`, `U+d55c`, `U+1f600`. In one pass `i` jumps 0 → 1 → 4 → 8, through `a` (1 byte), 한 (3 bytes) and 😀 (4 bytes). **Never advance with `add i 1`** — on the second round it would point into the middle of a character. If you only need the count, it is one line: `let n option u64 . be utf8.count_chars s .`

Counter-examples

> **Counter-example. Patching `none` with a replacement character and carrying on**
>
> Skipping a byte with `set i (add i 1)` when `next` fails lets invalid input through silently, and comparisons and filters above stand on a false premise. There is no error — it is **a silently**

wrong answer, and the character count quietly grows. Feed invalid input on purpose and see whether later processing runs although `is_valid` is `false`. The only fix is to stop at `none`.

Counter-example. Treating a byte offset as a character number

`decode s 1` is not "the second character" but "read at byte 1". Byte 1 of `"한글"` is the middle of a sequence, so it is `none`. If `is_valid` is `true` but one position gives `none`, that position is not a character boundary.

Counter-example. Treating `none` from `count_chars` as 0

`none` is not "no characters" but "invalid UTF-8". An empty slice is `some 0`. Merge the two and broken input can never be told apart from input with no characters. Taking it out with `some_value` without checking stops at run time with `E-VM-NONE`.

Counter-example. Slicing unvalidated input with `subslice` and assuming it is valid

Byte slicing always succeeds but can cut through the middle of a sequence. If the original is fine but only the piece fails `is_valid`, the cut was mid-character. Nobody warns at the cut, and it shows only when output breaks much later. To cut at character boundaries, cut only at positions `next` gave.

Cautions

- **`len s` is a byte count.** Only `count_chars` knows the character count. Measuring "at most 10 characters" with `len` goes wrong (`"한글"` has `len` 6 and 2 characters).
- **Advance the cursor with `next`.** Advancing by 1 reads continuation bytes as leads, giving `none` or an inflated count.
- **`none` from `decode` and from `next` judge different things.** `next` checks structure only, not value range (overlong, surrogates, upper limit). For strict iteration use both, or validate everything first with `is_valid`.
- **`count_chars` and `is_valid` are O(bytes).** Calling them in a loop condition every time makes the whole thing $O(n^2)$ — count once and carry the result.
- **It works in code points, not in characters as people count them.** Combining characters and emoji sequences count as several code points (one flag emoji is two).
- **Everything is `effects none`.** It can be called from a `fn`, a contract or `comptime`.

`utf16` — UTF-16 surrogate arithmetic

Source	`lib/utf16.low`
Layer	L0 — pure computation
Capabilities	none

Turns UTF-16 surrogate pairs into code points and back. Use it when exchanging values with worlds that hold strings as 16-bit units, such as the Windows API, Java and JavaScript. `slice u16` is a UTF-16 string, and one element is one unit.

```
use utf16 .

let c option u64 . be utf16.decode s 0 .
guard is_some c . else return 1 .
```

`0` is not “the first character” but **unit 0**. Every position in this module is a unit index, and advancing by character is done by `next` (two units for a pair).

Why surrogates exist. UTF-16 first tried to hold every character as “one character = one 16-bit unit”. When Unicode grew beyond 65,536 characters, the value range U+D800 … U+DFFF was left unused as characters, and two units from that range were combined to write large code points. 1024 high values × 1024 low values = 1,048,576, covering exactly U+10000 … U+10FFFF. So in UTF-16 **reading one unit and reading one character differ**.

Why it is a library. `slice u16` and `slice u32` already do reading, writing and indexing. No capability and no expression missing from the language is needed. So it became a library with no new type, word or builtin. The only real work left is **surrogate arithmetic** and how it fails. There is no UTF-32 library — one unit of `slice u32` is a code point, so all that is needed is one line of validation (`cp_valid`).

Design and boundaries

- **What it does** — converting between code points and UTF-16 units, and rejecting what is wrong. An unpaired high surrogate (D800 … DBFF not followed by a low), an unpaired low surrogate (DC00 … DFFF on its own), and an out-of-range code point (> U+10FFFF) give `none`.
- **It pairs with `utf8`.** `utf8` rejects surrogates inside UTF-8, so surrogates exist **only inside UTF-16 and only as pairs**. The two libraries keep that invariant together.
- **What it does not do** — normalisation (NFC, NFD), case, combining characters, byte order (BOM, big/little endian). `slice u16` is already in machine order.
- **Everything is `effects none`.** `put` is a `proc`, but it writes only into the buffer the caller handed over, not its own state, so it is `effects none` too.

A BMP code point (U+0000 … U+FFFF excluding the surrogate range) is one unit, value as is. U+10000 … U+10FFFF is two units — make 20 bits with `v = c - 0x10000` and split into the high unit `hi = 0xD800 + v/1024` and the low unit `lo = 0xDC00 + v%1024`. The reverse arithmetic is `c = 0x10000 + (hi - 0xD800) × 1024 + (lo - 0xDC00)`.

Unit value (decimal)	What it is	Meaning

55296 ⋯ 56319	High surrogate	The **front** piece of a two-unit character — a low must follow
56320 ⋯ 57343	Low surrogate	The **back** piece of a two-unit character — wrong if alone
Otherwise	An ordinary BMP unit	Its value is the code point

Table 50.1 — What a single unit is

The code point upper limit is 1114111 (U+10FFFF). Surrogate values themselves are not code points, and `cp_valid` rejects them.

Ops at a glance

op	Signature	When it cannot
is_high	`fn (u u64) → bool`	never fails
is_low	`fn (u u64) → bool`	never fails
cp_valid	`fn (c u64) → bool`	never fails (false is the answer)
units	`fn (c u64) → u64`	never fails (1 or 2)
unit_hi	`fn (c u64) → option u64`	`none` for an invalid code point
unit_lo	`fn (c u64) → option u64`	`none` for an invalid code point or BMP (one unit)
put	`proc (dst mut slice u16, at u64, c u64) → option u64`	`none` for an invalid code point or no room
decode	`fn (s slice u16, at u64) → option u64`	`none` if out of range or unpaired
next	`fn (s slice u16, at u64) → option u64`	`none` at the end or at a broken place
count_chars	`fn (s slice u16) → option u64`	`none` on meeting a broken place
is_valid	`fn (s slice u16) → bool`	never fails

Table 50.2 — Ops of `utf16`

The most used in practice are `put` (writing) and `decode` + `next` (iteration).

Ops in detail

- **`is_high` · `is_low`** — is a unit's value a high (D800 … DBFF) or low (DC00 … DFFF) surrogate. A judgement by value range alone.
- **`cp_valid`** — is it valid as a code point. `false` if `c > 1114111` or in the surrogate range (55296 … 57343). The gate that filters numbers from outside before writing.
- **`units`** — how many units a code point takes. 2 if `c ≥ 65536`, otherwise 1. Use it to size buffers in advance. **It does not check validity.**
- **`unit_hi`** — the high unit. `some c` for BMP, `some (0xD800 + v/1024)` for two units, `none` if invalid.
- **`unit_lo`** — the low unit. `none` for BMP — here meaning "there is only one unit", not an error. `some (0xDC00 + v%1024)` for two units. `none` also for an invalid code point.
- **`put`** — writes the code point into `dst` from `at` and returns **the next position to write** (`some (at+1)` or `some (at+2)`). It is on the same position axis as `decode` and `next`. If you need how many units

were written, `units c` answers. If invalid or `at + units c > len dst`, `none`, and **not a single unit is written**. It makes no memory itself, so the place to write comes from outside.

- **`decode`** — reads one code point at `at` in `s`. An ordinary unit gives its value; a high surrogate checks that the next unit is a low and combines them. `none` if `at ≥ len s`, a low comes alone, a high is not followed by a low, or it ends with a high (truncated). A pair is two units, so it takes the whole slice. `at` must be a character's **start** unit.
- **`next`** — the position of the next code point. `decode` must succeed; starting at a high surrogate gives `some (at+2)`, otherwise `some (at+1)`. It spares the caller counting two units for a pair.
- **`count_chars`** — walks everything and counts code points. On a broken place it **gives no count** — better no count than a wrong one. O(units).
- **`is_valid`** — whether `count_chars` is `some`. Half a pair looks invalid on its own, so validity is not asked in parts.

Using it

Writing calls `put` and continues from the returned next position. Reading reads values with `decode` and advances with `next`. Either way the caller holds the cursor.

```
module ex_utf16 .

use utf16 as u .

proc round_trip input buf mut slice u16 . . output u64 . effects none . do
  guard ge (len buf) 4 . else return 90 .
  rem '가' (U+AC00) --- BMP, so one unit. put returns the next position to write
  let a option u64 . be u.put buf 0 44032 .
  guard is_some a . else return 1 .
  guard eq (some_value a) 1 . else return 2 .

  rem U+1F4A9 --- outside the BMP, so split into two units (a surrogate pair)
  let b option u64 . be u.put buf (some_value a) 128169 .
  guard is_some b . else return 3 .
  guard eq (some_value b) 3 . else return 4 .
  guard eq (index buf 1) 55357 . else return 5 .
  guard eq (index buf 2) 56489 . else return 6 .

  rem read it back as one --- decode joins the pair
  let rb option u64 . be u.decode buf 1 .
  guard is_some rb . else return 7 .
  guard eq (some_value rb) 128169 . else return 8 .

  rem 'A' + emoji (pair) + '가' = four units, three characters
  set (index buf 0) 65 .
  set (index buf 3) 44032 .
  let n option u64 . be u.count_chars (subslice buf 0 4) .
  guard is_some n . else return 9 .
  guard eq (some_value n) 3 . else return 10 .
  return 42 .
end
```

The high unit `buf[1]` is 55357 (0xD83D) and the low unit `buf[2]` is 56489 (0xDCA9). Iterate with `next`.

```
var i u64 be 0 .
while lt i (len s) . do
  let c option u64 . be u.decode s i .
  guard is_some c . else return 80 .
  let nx option u64 . be u.next s i .
  guard is_some nx . else return 81 .
  set i (some_value nx) .
end
```

+1 for BMP, +2 for a pair — you do not count. `none` means stop (whether it is the end or breakage is told apart by measuring `is_valid` first).

Counter-examples

> **Counter-example. Trying to pass unpaired input through `decode`**
>
> ```
> set (index buf 0) 55357 .          rem ✗ after the high 0xD83D
> set (index buf 1) 65 .             rem   comes 'A', not a low
> let a option u64 . be u.decode buf 0 .
> ```
>
> It is `none`. So is a low surrogate alone (`decode buf 2`, value 56489) or ending with a high surrogate (`decode (subslice buf 0 4) 3`). All three are caught right at `guard is_some`. Letting them through silently would make everything after wrong, so **rejection is the value of this library**. Intact pairs still pass.

> **Counter-example. Putting a two-unit character into the last unit**
>
> `u.put buf (sub (len buf) 1) 128169` is `none` and **that unit is not touched**. No need to worry about half a write.

> **Counter-example. Trying to use a surrogate value as a code point**
>
> `u.cp_valid 55357` is `false`, so `put`, `unit_hi` and `unit_lo` all give `none`. Taking these `none`s out with `some_value` without checking stops at run time with `E-VM-NONE` — it translates, so `guard` comes first.

Cautions

- **`len s` is a unit count.** Only `count_chars` knows the character count. One emoji takes two units.
- **Do not cut through a pair.** Splitting between high and low with `subslice` makes both pieces invalid. Cut only at positions `next` gave.
- **Advance the cursor with `next`.** Advancing by 1 reads a pair's low unit as a character start, giving `none` or an inflated count.
- **Costs.** `decode`, `next` and `put` are O(1); `count_chars` and `is_valid` are O(units). Do not count in a loop condition every time.
- **`none` from `unit_lo` has two overlapping meanings** (invalid code point · BMP). If you need to tell them apart, ask `cp_valid` first. When combining, `put` is better, since its failure collapses into one `none`.
- **Byte serialisation is outside.** Turning **bytes** from a file or network into a `slice u16` is the caller's job.

`unicode` — Unicode property tables

Source	`lib/unicode.low`
Layer	L0 — pure computation
Capabilities	none

This is where you ask “is this character a letter, a number, a space?”. With ASCII only, you could answer by hand: `ge c 97` and `le c 122` means lowercase. The moment `'한'`, `'あ'` or `'Ω'` arrives, that approach collapses. The code points that are “letters” in Unicode span **658 scattered ranges**, impossible to write by hand. This module carries those range tables **extracted mechanically from the Unicode 15.1.0 source** and answers by binary search.

```
use unicode .

proc ident_start input cp u64 . output bool . effects none . do
  if unicode.is_letter cp . do return true . end
  return eq cp 95 .
end
```

`is_letter 54620` (`'한'`) is `true` and `is_letter 128512` (😀) is `false` — emoji are symbols, not letters. **The input is a code point (`u64`), not a byte.** Getting code points out of UTF-8 byte strings is `utf8`’s job, and the two are used together.

Tables are data — written as code, lies creep in. Without this module everyone writes letter tests by hand in their own programs, and those tests are almost always wrong. The problem is how they are wrong. A missed range comes out not as an error but as `false` — **a silently wrong answer**. A parser cuts a token at that character, a search misses that word, and no warning appears anywhere. Display width (`term`’s `cp_width`) had a usable default of 1 even when wrong, but here there is no such default.

Design and boundaries

- **Tables are carried as one fixed-width hex string.** One range is `<start 6 digits><end 6 digits>` = 12 bytes. The position of range `n` is simply `n × 12`, so binary search works without decoding. No array to expand, no initialisation code, no allocation, so the whole layer stays `effects none` and is verified inside the VM/native cross-check.
- **The range is U+0000 … U+2FFFF.** Above that is still hardly used. **It does not pretend to carry what it does not** — everything from U+30000 up answers “no property”.
- **Subcategories are split when something needs to ask.** Lu, Ll and Nd were split when regex `\p{Lu}` created a place to ask, and the rest followed. All subcategories of L and N now exist (L = Lu+Ll+Lt+Lm+Lo, N = Nd+Nl+No). Subcategories of P, Z and M are not split yet.
- **Tables can be re-extracted.** The development repository’s extraction script produces the tables and checks every time that **the tables carried match the extraction byte for byte**. A table pasted by hand once becomes, from that moment, an unverifiable lump of constants. This check has caught something — an old document claimed `is_space` included tab and newline, and the table did not.

- **Conversion is pairs, not ranges.** Categories ask "is it in this range", conversion asks "**where** does it go". So `tab_toupper` and `tab_tolower` use the same fixed width but carry `<from><to>` pairs. **Only one-to-one** mappings are carried, and that limit is not hidden (`is_special_case`).
- **Not built** — normalisation (NFC, NFD), composition, subcategories of P, Z and M, character names, full case folding (conversions that change length, like ß → ss). Each needs yet another table, and tables are not free.

Tables are `slice u8` returned by `fn`s (string literals — no copy, no allocation), and there is no struct or state. Tables keep four invariants — a range is 12 lowercase hex bytes, sorted by start ascending, no overlaps (adjacent ranges merged), and **end inclusive** (`start ≤ cp ≤ end`). If you build a table yourself for `has_cp`, keep sorting and no overlap for binary search to hold.

Ops at a glance

op	What it does
`tab_letter` · `tab_number`	Letter (L) table, 658 ranges · number (N) table, 137 ranges
`tab_punct` · `tab_space` · `tab_mark`	Punctuation (P) · space (Z, no tab or newline) · mark (M) tables
`tab_zerowidth`	**Zero width** (Mn, Me, Cf) table, 354 ranges
`tab_upper` · `tab_lower` · `tab_digit`	Uppercase (Lu) 646 · lowercase (Ll) 658 · decimal digit (Nd) 64 ranges
`tab_title` · `tab_modifier` · `tab_other_letter`	Lt 10 · Lm 71 · Lo 509 ranges
`tab_numletter` · `tab_numother`	Nl 12 · No 72 ranges
`tab_toupper` · `tab_tolower`	Case **pair** tables, 1423 · 1432 pairs
`tab_casespecial`	103 code points that cannot be converted one-to-one
`range_count`	Number of ranges in a table (`len tab / 12`)
`has_cp`	Is this code point in the table — binary search
`is_letter` · `is_number` · `is_punct` · `is_space` · `is_mark`	Predicates for each table
`is_alnum`	Letter or number
`is_upper` · `is_lower` · `is_digit`	Lu · Ll · **decimal** digit (Nd)?
`is_zerowidth`	Zero width (combining or format character)?
`is_title` · `is_modifier` · `is_other_letter` · `is_numletter` · `is_numother`	Lt · Lm · Lo · Nl · No?
`to_upper` · `to_lower`	To uppercase · lowercase (**unchanged** if no mapping)
`is_special_case`	A code point that cannot be converted one-to-one (such as ß)?

Table 50.1 — Ops of `unicode`

Ops in detail

- **`tab_*`** — no parameters; each returns a read-only view valid for the life of the program. You rarely use them directly — mainly to run several tables in one loop or to see sizes with `range_count`.
- **`range_count tab`** — `len tab / 12`. Passing a slice that is not a table gives a meaningless number (it does not check).
- **`has_cp tab cp`** — the core. Cost is O(log n) comparisons, each reading two 6-digit hex numbers. No allocation, no buffer. **If the table is broken** (non-hex bytes), it returns `false` — it invents nothing, which also means it passes a broken table silently.
- **`is_letter` and friends** — thin wrappers over `has_cp <table> cp`. Any `u64` gets an answer (`false` out of range). `is_alnum` exists separately as the combination most used for identifier checks.
- **`is_digit` is narrower than `is_number`.** The Roman numeral Ⅶ (U+2166) and the fraction ½ are N but not Nd. Computing digit values (`d = cp - '0'`) requires Nd.
- **Korean is `false` for both `is_upper` and `is_lower`** (Lo — characters without case). Yet `is_letter` is `true`. “Not uppercase” does not mean “not a letter”.
- **`is_zerowidth`** — `true` for Mn (nonspacing marks), Me (enclosing marks) and Cf (format characters). These code points take no cell of their own; they sit on the previous character. `term`’s width computation and cluster counting come from **the same table**, so “width right but cursor movement wrong” cannot happen. `is_mark` is the whole M category (including Mc, which does take a cell), so use `is_zerowidth` to count width.
- **`to_upper` · `to_lower`** — if nothing changes they return the input as is (not `none`). Most code points in the world have no case, and making callers unwrap an `option` every time would fill call sites with `some_value`. **Only one-to-one** — the uppercase of German ß (U+00DF) is the two letters `"SS"`, which cannot be written as a pair, so `to_upper 223` returns 223 as is. `is_special_case` names such code points separately. Round trips do not always hold either (Turkish `ı`, Greek final sigma). Case-insensitive comparison is folding, which does not exist yet.

Using it

Counting words (runs of letters and numbers) in a UTF-8 string:

```
module wordcount .

use unicode .
use utf8 .

export proc count_words input s slice u8 . output option u64 . do
  var off u64 be 0 .
  var words u64 be 0 .
  var inword bool be false .
  while lt off (len s) . do
    let cp option u64 . be utf8.decode s off .
    guard is_some cp . else return none .
    let n u64 be utf8.seq_len (index s off) .
    guard gt n 0 . else return none .
    if unicode.is_alnum (some_value cp) . do
      if eq inword false . do set words (add words 1) . end
      set inword true .
```

```
    end
    if eq (unicode.is_alnum (some_value cp)) false . do
      set inword false .
    end
    set off (add off n) .
  end
  return some words .
end
```

`"한글 word 123"` gives 3 — a test that knows only ASCII cannot count `'한글'`. Regex `\p{L}` and `\P{L}` use exactly these tables (`regex`). If a regular expression is enough, use it; where only one code point needs asking, calling this module directly is much cheaper.

Counter-examples

Counter-example. Passing a byte as is

```
rem ✘ index gives a byte. The first byte of '한' is 0xED
if unicode.is_letter (widen u64 (index s 0)) . do … end
```

0xED (237) is the code point U+00ED (í) — judged a letter by chance, but **not the character being asked about**. When handling UTF-8, always go through `utf8.decode`.

Counter-example. Using `is_mark` while counting width

It counts Mc (combining marks that take a cell) as 0 too. Width needs `is_zerowidth` — and `term`'s `cp_width` already handles width anyway.

Counter-example. Assuming `to_upper` always converts

ß comes back unchanged. If precision matters, ask `is_special_case` first; expanding to two or more characters is the caller's decision.

Counter-example. Computing digit values with `is_number`

```
rem ✘ Ⅶ (U+2166) passes too, and cp - 48 is meaningless
if unicode.is_number cp . do set v (add (mul v 10) (sub cp 48)) . end
```

To accept decimal digits only, use `is_digit`.

Counter-example. Treating `false` from `is_upper` as lowercase

Korean, Chinese and Arabic characters are `false` for both. That test classifies most of the world's characters as lowercase. Ask `is_lower` directly.

Cautions

- **Out of range is silently `false`.** A program handling U+30000 and above must know this module cannot answer.
- **`false` from `has_cp` does not distinguish "not in it" from "the table is broken".** Built-in tables are always intact. If you build a table, check that `mod (len tab) 12` is 0 before passing it.

- **Tables follow Unicode 15.1.0.** When Unicode updates, re-extract the tables rather than editing them by hand.
- **Binary size.** All tables go in as strings (tens of KB). If that matters on a small machine, a module carrying only the needed tables is the right answer.
- **`is_space` is exactly category Z — tab, newline and CR are not in it.** In Unicode those are control characters (Cc). If you want regex `\s` behaviour, add them at the call site. They were not mixed in because this table is used as is for `\p{Z}`.

codec — hex and base64

Source	`lib/codec.low`
Layer	L0 — pure computation (the caller's buffer)
Capabilities	none

Turns bytes **into hex or base64 text and back**. Many places let only text through — a line of a configuration file, a log, a URL, a mail body, screen output. Putting raw bytes there lets zero and control bytes break the channel. Hex (one byte as two characters `0-9 a-f`) and base64 (arbitrary bytes as 64 characters `A-Z a-z 0-9 + /`) are **wrappings for carrying arbitrary bytes using only safe characters**. Hex is easy to read but exactly doubles the size; base64 is hard to read but ends at about 4/3. Hex is usual for eyeballing hashes, base64 for carrying chunks.

```
use codec as c .

let n option u64 . be c.hex_enc "abc" dst .
guard is_some n . else return 1 .
```

`dst` is not made by this module but is **an output place the caller sets aside in advance** (at least 6 bytes here). `n` is **the number of bytes actually written**, and the result is `dst[0..n)`.

A wrapping keeps no secrets — anyone can undo it. If secrecy is needed, use `aead`, which seals (chapter 33). This module is pure computation from bytes to bytes, using only `index`, `set`, `shl`, `shr`, `bit_and`, `bit_or`, `guard` and `while`, and no builtin was added.

Design and boundaries

- **Invalid input is answered with a value — one kind, `none`.** Odd-length hex, non-hex characters, non-base64 characters, misplaced padding and **a too-small output buffer** all give `none`. It neither stops nor truncates — truncating would put every layer after on the false premise "all written". Success is `some <bytes written>`, and the caller gets the exact result with `subslice`.
- **Hex encoding emits lowercase.** Decoding accepts both cases — liberal in what it accepts, conservative in what it sends.
- **base64 uses the standard alphabet** (`A-Z a-z 0-9 + /`) and `=` padding. base64 writes input 3 bytes at a time as 4 characters, and pads a short last group with `=` so the length is always a multiple of 4.
- **Not built** — the URL-safe variant (`-_`) and line wrapping. If needed they will be built as **separate ops** — a mode argument on one op is itself confusion.
- **The output buffer belongs to the caller.** Only the caller knows the cost and policy of providing a buffer. So it is `effects none` with no hidden allocation.

op	`dst` size needed	Success returns
hex_enc	`2 × len src`	`some (2 × len src)`
hex_dec	`len src / 2`	`some (len src / 2)`
b64_enc	`4 × ⌈len src / 3⌉`	`some (4 × ⌈len src / 3⌉)`
b64_dec	`3 × (len src / 4) − padding count`	`some <bytes written>`

Table 50.1 — Output room needed

If `dst` is smaller than the formula the result is always `none`; if larger it does not matter — it writes from the front and returns the count. Empty input succeeds (`hex_enc ""`, `b64_enc ""` and `b64_dec ""` all give `some 0`).

Ops at a glance

op	Signature	When it cannot
`hex_digit`	`fn (v u64) → u8`	never fails (assumes 0 … 15)
`hex_val`	`fn (c u8) → option u64`	`none` outside `0-9 a-f A-F`
`hex_enc`	`proc (src slice u8, dst mut slice u8) → option u64`	`none` if `dst` too small
`hex_dec`	`proc (src slice u8, dst mut slice u8) → option u64`	`none` for odd length · non-hex characters · `dst` too small
`b64_digit`	`fn (v u64) → u8`	never fails (assumes 0 … 63)
`b64_val`	`fn (c u8) → option u64`	`none` outside the alphabet (`=` too)
`b64_enc`	`proc (src slice u8, dst mut slice u8) → option u64`	`none` if `dst` too small
`b64_dec`	`proc (src slice u8, dst mut slice u8) → option u64`	`none` if length not a multiple of 4 · outside the alphabet · misplaced padding · `dst` too small

Table 50.2 — Ops of `codec`

Usually only `hex_enc`, `hex_dec`, `b64_enc` and `b64_dec` are used; the other four are single-character parts.

Ops in detail

Every conversion takes `src` (bytes to read) and `dst` (place to write) separately. There is no in-place conversion because encoding grows the result, and overwriting the original would trample bytes not yet read. **The length of `src` is the whole input** — no end marker is sought, only `len src` is trusted, so when passing a result held in a larger buffer back in, cut it exactly with `subslice`. The result length is not the length of `dst` but the `n` in the returned `some n`.

- **`hex_digit v`** — a nibble (0 … 15) as a lowercase hex character. `'0'+v` if `v < 10`, otherwise `'a'+(v−10)`. It does not check the range; the caller guarantees it.
- **`hex_val c`** — the value of one hex character. Accepts both cases; `none` otherwise.
- **`hex_enc src dst`** — first checks `2×len src ≤ len dst`, and if short writes nothing and gives `none`.
- **`hex_dec src dst`** — needs even length, all characters hex, and `len src / 2 ≤ len dst`. A non-hex character is found **while scanning**, so bytes before it may already be written to `dst`. On `none`, treat the content of `dst` as undefined.
- **`b64_digit v`** — the `v`-th (0 … 63) character of the standard alphabet. 0 … 25 are `A-Z`, 26 … 51 `a-z`, 52 … 61 `0-9`, 62 `+`, 63 `/`. base64 maps 6-bit pieces, not bytes, to characters.
- **`b64_val c`** — the value of one base64 character. **`=` is `none` too** — padding is not a character but structure handled by the decoder body.
- **`b64_enc src dst`** — writes including `=` padding. A 1-byte tail gives `XX==`, a 2-byte tail `XXX=`.

- **`b64_dec src dst`** — needs a length that is a multiple of 4, every character in the alphabet or valid padding, and enough `dst`. `=` may appear only in **the last group**, and if the third position is `=` the fourth must be too. Success gives the exact length reduced by the padding. On `none` the content of `dst` is undefined.

Using it

Check buffer sizes first, convert, and cut by the returned count.

```
module ex_codec .

use codec as c .

proc hex_roundtrip input enc mut slice u8 . . input dec mut slice u8 . . output u64 . effects
none . do
  guard ge (len enc) 6 . else return 90 .
  guard ge (len dec) 3 . else return 91 .
  let en option u64 . be c.hex_enc "abc" enc .
  guard is_some en . else return 1 .
  guard eq (some_value en) 6 . else return 2 .
  rem the count written is the boundary --- passing all of enc makes trailing garbage input
  let dn option u64 . be c.hex_dec (subslice enc 0 6) dec .
  guard is_some dn . else return 3 .
  guard eq (some_value dn) 3 . else return 4 .
  guard eq (index dec 0) 97 . else return 5 .
  rem uppercase is accepted too: "4A" → 74
  let up option u64 . be c.hex_dec "4A" dec .
  guard is_some up . else return 6 .
  guard eq (index dec 0) 74 . else return 7 .
  return 42 .
end

proc b64_roundtrip input enc mut slice u8 . . input dec mut slice u8 . . output u64 . effects
none . do
  rem 2 bytes make one group (4 characters) with one = at the end: "aGk="
  guard ge (len enc) 4 . else return 90 .
  guard ge (len dec) 2 . else return 91 .
  let e option u64 . be c.b64_enc "hi" enc .
  guard is_some e . else return 1 .
  guard eq (some_value e) 4 . else return 2 .
  let d option u64 . be c.b64_dec (subslice enc 0 4) dec .
  guard is_some d . else return 3 .
  guard eq (some_value d) 2 . else return 4 .
  return 42 .
end
```

Counter-examples

None of these stop; all give `none`. The symptom is always “the result is `none` and `guard` falls to `else`”.

Input	Result	Usual cause, and `dst`

`c.hex_dec "abc" dst`	none	Odd length — caught by the length check first, `dst` unchanged
`c.hex_dec "zz" dst`	none	Non-hex character — found while scanning, so the front of `dst` may be overwritten
`c.hex_enc "abcdefgh" dst3`	none	16 needed in 3 bytes — not a byte is written
`c.b64_dec "abc" dst`	none	Length not a multiple of 4 — often `enc` passed whole instead of cut with `subslice`
`c.b64_dec "a?cd" dst`	none	Outside the alphabet — mostly URL-safe `-`, `_` or newlines mixed in
`c.b64_dec "aG==YWJj" dst`	none	Padding in the middle — what you get by simply concatenating two base64 strings

Table 50.3 — Inputs that commonly go wrong

Cautions

- **If `none` keeps coming, suspect the buffer size first.** The cause of "the code is right but always `none`" is almost always a `dst` smaller than the formula. Allocating generously costs nothing.
- **On `none`, do not use `dst` as the result.** Encoders check size first so `dst` is unchanged on failure, but decoders write while scanning, so the front may be overwritten. Only `dst[0..n)` of a successful `some n` is the result.
- **Do not discard the returned count and use all of `dst` as the result.** Old bytes left behind mix in.
- **Do not overlap the buffers of encoding and decoding.** If `src` and `dst` overlap, output overwrites input not yet read. Overlap is not checked, and the result is silently wrong.
- **Do not handle padding by hand.** Asking `b64_val` about `=` gives `none`. Leave it to `b64_dec`.
- **URL-safe base64 and multi-line base64 are not accepted** — unsupported, not a defect. The caller strips newlines before passing it in.
- **Linear cost, free reentrancy.** Everything is linear in input length with no state, so calls from many places at once do not interfere.

`regex` — Pike VM regular expressions without backtracking

Source	`lib/regex.low`
Layer	L0 — pure computation (the caller's scratch)
Capabilities	none

Checks and finds **what shape** a string has. Use it for input validation (did only digits arrive), finding patterns in logs, and simple tokenising. A pattern is **compiled** once into a program, and the same program matches many inputs.

```
use regex as rx .

let n option u64 . be rx.compile "a{2,4}b" prog st .
guard is_some n . else return 1 .
let m option u64 . be rx.match_at prog "aaab" 0 cl nl mk .
```

The program (`slice u64`), the compiler state and the matcher's workspace are all the caller's slices. So it is `effects none` with no hidden allocation, reentrancy is free, and it runs inside the VM/native cross-check.

Why a Pike VM is safe. Common regex engines **backtrack**. At a fork (`|`, `*`) they follow one path to the end and come back to try another when stuck. When forks nest, paths to try multiply 2, 4, 8…, and one pattern like `(a*)*b` can take minutes on a few dozen bytes of input — that is ReDoS (regular expression denial of service). A Pike VM never goes back. It treats the pattern as an NFA and holds every "place it could be now" in a **thread list** (not OS threads, but values recording which instruction of the program it stands on), advancing the whole list **one step at a time together** for each input byte. The same place is held only once, so the list never exceeds the number of instructions. Hence **exponential blow-up is impossible in principle** for any pattern and input — one match from a fixed start is O(input × program).

Design and boundaries

Syntax accepted

- Literals, `.` (except newline), `* + ?` (greedy), `[a-z]` and `[^…]`, `|`, `(…)`, `^ $` (string boundaries), escapes `\d \w \s \n \t \r` and literalising special characters (`\.` and the like).
- Inside classes, the literals `\n \t \r \\ \] \-`, unions of the sets `\d \w \s`, ranges `a-z` and negation `[^…]`.
- Repetition `{m}`, `{m,}`, `{m,n}` (greedy) — **expanded at compile time**. A run-time counter would give each thread private state, breaking the invariant "a thread is one instruction position" and collapsing deduplication. The upper limit is 64.
- Capture groups `match_caps`, `\X` consuming one UTF-8 character, and Unicode properties `\p{…}` and `\P{…}` (`L N P Z M` and `Lu Ll Lt Lm Lo Nd Nl No`).

Properties look back after consuming. Property sets have hundreds to thousands of ranges, so expanding them into a byte automaton yields thousands of instructions — unmanageable in a design where the program buffer belongs to the caller. Instead, **after** consuming one code point it looks back to check the property. UTF-8 is self-synchronising and can be read backwards, and the check consumes no bytes, so the grounds for linearity stay. The tables are held by `unicode`.

Unknown property names are **rejected** — answering with a broader table would be a silently wrong match.

Not built — properties inside classes (`[\p{L}0-9]`; classes are byte bitmaps), case-insensitive matching (`(?i)`; needs folding tables), and backreferences and lookaround (closer to never being built — linearity dies the moment they arrive). Invalid patterns and short buffers are all rejected **as values** (`none`).

Data structures

A whole program lives in one **program buffer** (`slice u64`). `prog[0]` is the instruction count, `prog[1]` the class count, followed by instructions of three words each, `[op a b]`. **Code grows from the front, and character class bitmaps (256 bits = 4 words) grow backwards from the tail** — when the two meet, compilation gives `none`. The bitmap of class k is at `prog[len - 4·(k+1) .. len - 4·k)`.

Code · name	a · b	Meaning
0 CHAR	byte value	consume that byte and advance
1 ANY	—	consume any byte except newline (10)
2 CLASS	class number k	consume a byte in bitmap k
3 SPLIT	branch 1 · branch 2	run both branches (branch 1 first = greedy)
4 JMP	target	unconditional jump
5 MATCH	—	reaching here is a match
6 BOL · 7 EOL	—	`^` passes only at position 0, `$` only at `len s`
8 SAVE	slot number	record a capture position (consumes no byte)
9 CPPROP	property code	is the code point **just consumed** of that property (consumes no byte)

Table 50.1 — Instruction codes

Property codes are 0=L · 1=N · 2=P · 3=Z · 4=M · 5=Lu · 6=Ll · 7=Nd · 8=Lt · 9=Lm · 10=Lo · 11=Nl · 12=No, and **negation adds 16** (`\P{L}` = 16). Because the negation offset was generous, the encoding did not change by a single bit when the tables grew from 8 to 13.

Compiler state `st` is four slots (instruction count · class count · pattern position · failure flag), initialised by `compile`, so only length 4 or more must be ensured. **Matcher scratch** is the current and next thread lists `clist` and `nlist`, the deduplication stamps `marks`, and the start position arrays `cstart` and `nstart` that `search` also takes. All are `slice u64`, each **at least the instruction count long**.

Ops at a glance

op	Signature	When it cannot
compile	`proc (pat slice u8, prog mut slice u64, st mut slice u64) → option u64`	`none` on syntax error or short buffer

match_at	`proc (prog slice u64, s slice u8, at u64, clist, nlist, marks) → option u64`	`none` for no match · empty program · short scratch
find	`proc (prog slice u64, s slice u8, clist, nlist, marks) → option u64`	`none` if no match anywhere
search	`proc (prog slice u64, s slice u8, clist, nlist, marks, cstart, nstart) → option u64`	`none` for no match anywhere · empty program · short scratch
test_at	`proc (prog slice u64, s slice u8, clist, nlist, marks) → bool`	never fails (false is the answer)
class_has	`fn (prog slice u64, k u64, b u64) → bool`	never fails

Table 50.2 — Ops of `regex` — all `effects none`

Names starting with `rx_` are compiler and matcher internals, invisible from outside. Scratch parameters are all `mut slice u64`.

Ops in detail

- **`compile pat prog st`** — needs `len prog ≥ 8` and `len st ≥ 4`. Success gives `some <instruction count>`, which is the minimum length for matcher scratch. `none` for an unclosed group `(ab`, `) | * + ?` in atom position, an unclosed `[…]`, a reversed range `[z-a]`, a lone `\` at the end, bad counts (`a{4,2}`, `a{}`, `a{2`, over 64), or code meeting bitmaps (buffer too small). The empty pattern is valid — it compiles to one `MATCH` and matches empty anywhere.
- **`match_at prog s at clist nlist marks`** — gives **the longest end** (greedy) of a match starting at `at` as `some end` (exclusive). The start is part of the question, so it takes `at`. `clist` is the thread list at the current position, `nlist` the list at the next (the two swap each byte), and `marks` stamps "this instruction was already added at this position". Without stamps the same place enters several times and the list-length bound breaks. An empty match is a match (`some at`). Cost is O(`len s` × instructions).
- **`find`** — **the leftmost position where a match starts**. It restarts `match_at` for `at = 0, 1, …, len s`, so the bound is $O(n^2 \cdot m)$, but each attempt is linear, so there is no exponential blow-up. If you need the end, call `match_at` once more from the returned start.
- **`search`** — answers the same question as `find` **in one pass**. Removing the restart loop means each thread must carry "where it started", hence two more scratch slices `cstart` and `nstart`. A start thread is added at each position only while no match has been found yet, so cost is O(input × instructions), and since a further-left start always wins, leftmost priority holds. It gives **the same answer** as `find` — the development repository's tests compare the two.
- **`test_at`** — is there a match anywhere. Use it when you need only yes or no, not a position.
- **`class_has prog k b`** — does class `k` of the compiled program contain byte `b`. The bitmap lives in the tail of the program buffer, so it takes `prog`. The range of `k` is not checked.

Using it

Escapes are two layers deep. Lowent string literal escapes are a closed set of fourteen (chapter 3), and `\d` or `\w` are not among them (others are `E-STR-ESCAPE`). To write regex `\d` in source, write **`"\\d"`** — the literal turns `\\` into one backslash, and the regex compiler reads those two bytes `\d` as the digit class.

```
module ex_regex .

use regex as rx .

proc demo input prog mut slice u64 . . input st mut slice u64 . .
  input cl mut slice u64 . . input nl mut slice u64 . . input mk mut slice u64 . .
  output u64 . effects none .
do
  guard ge (len prog) 32 . else return 90 .

  rem "ab.d*" on "abcdddx": . consumes c, greedy d* consumes ddd, end = 6
  let n option u64 . be rx.compile "ab.d*" prog st .
  guard is_some n . else return 1 .
  let m option u64 . be rx.match_at prog "abcdddx" 0 cl nl mk .
  guard is_some m . else return 2 .
  guard eq (some_value m) 6 . else return 3 .

  rem as a regex this is [a-c]+z\d --- in source, "\\d"
  let n2 option u64 . be rx.compile "[a-c]+z\\d" prog st .
  guard is_some n2 . else return 4 .
  let m2 option u64 . be rx.match_at prog "abz7" 0 cl nl mk .
  guard is_some m2 . else return 5 .
  guard eq (some_value m2) 4 . else return 6 .

  rem "ab$" only at the end --- find gives the leftmost start
  let n3 option u64 . be rx.compile "ab$" prog st .
  guard is_some n3 . else return 7 .
  let f option u64 . be rx.find prog "xxab" cl nl mk .
  guard is_some f . else return 8 .
  guard eq (some_value f) 2 . else return 9 .
  let g option u64 . be rx.find prog "abx" cl nl mk .
  guard eq (is_some g) false . else return 10 .
  return 42 .
end
```

`search` and counted repetition have the same shape.

```
proc demo_search input prog mut slice u64 . . input st mut slice u64 . .
  input cl mut slice u64 . . input nl mut slice u64 . . input mk mut slice u64 . .
  input cs mut slice u64 . . input ns mut slice u64 . .
  output u64 . effects none .
do
  guard ge (len prog) 32 . else return 90 .
  let n option u64 . be rx.compile "b+c" prog st .
  guard is_some n . else return 1 .
  let a option u64 . be rx.find prog "xxbbbc" cl nl mk .
  guard is_some a . else return 2 .
  let b option u64 . be rx.search prog "xxbbbc" cl nl mk cs ns .
  guard is_some b . else return 3 .
  guard eq (some_value a) (some_value b) . else return 4 .
  guard eq (some_value b) 2 . else return 5 .

  let n3 option u64 . be rx.compile "a{2,4}b" prog st .
  guard is_some n3 . else return 9 .
  let m3 option u64 . be rx.match_at prog "aaab" 0 cl nl mk .
  guard is_some m3 . else return 10 .
  guard eq (some_value m3) 4 . else return 11 .
  rem below the lower bound is not a match
  let f option u64 . be rx.match_at prog "ab" 0 cl nl mk .
```

```
  guard eq (is_some f) false . else return 12 .
  return 42 .
end
```

The caller provides, for example, `prog` of 32 slots, `st` of 4, and `cl`, `nl`, `mk` (and `cs`, `ns` for `search`) of 32 each — anything above the instruction count `compile` returned. `{m,n}` expands, so the instruction count grows with the repetition count.

A backtracking engine would wander through 2^{64} paths giving `(a*)*b` sixty-four `a`s, but a Pike VM finishes linearly with its thread list.

```
let n option u64 . be rx.compile "(a*)*b" prog st .
guard is_some n . else return 1 .
let f option u64 . be rx.match_at prog
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" 0 cl nl mk .
guard eq (is_some f) false . else return 2 .
```

Counter-examples

Call	Result
`rx.compile "(ab" prog st`	none — unclosed group
`rx.compile "*a" prog st`	none — repetition with nothing before it
`rx.compile "abcdefghij" tiny st` (`tiny` of 8 slots)	none — code meets bitmaps
`rx.match_at prog s 0 short short short`	none — scratch shorter than the instruction count
`rx.compile "a{4,2}" prog st` · `"a{}"` · `"a{2"` · `"a{65}"`	none — bad counts (the limit of 64 stops blow-up, since repetition expands)
`rx.compile "\d" prog st`	`E-STR-ESCAPE` before running — write `"\\d"`

Table 50.3 — What is rejected as a value, and what at translation

Cautions

- **Pass `prog` to matching at the same length as at compilation.** Class bitmaps live at the buffer's tail, addressed relative to `len prog`. Passing a `subslice` keeps the instructions but misaligns the bitmap addresses, so `[…]` and `\d\w\s` silently look at the wrong bytes. Do not cut the program; carry it whole.
- **One program buffer, one pattern.** Compiling again overwrites the previous program. To alternate two patterns, provide two buffers.
- **`find` or `search` is a choice of cost.** The answers are the same. For long inputs and frequent searching use `search`; to save scratch on small inputs use `find`.
- **`marks` re-initialises itself every call.** Nothing needs preserving between calls.
- **Watch for empty matches.** Patterns like `a*` succeed with length 0 anywhere, and `find` always gives `some 0` for them. If you need "consumed at least one character", compare the end with the start.
- **The matching unit is the byte.** On UTF-8 input, `.` and `[^…]` consume one byte, not one code point. Ask for one code point with `\X`, and for Unicode categories with `\p{…}`.

`term` — terminal renderer (the pure half)

Source	`lib/term.low`
Layer	L0 — pure computation (the caller's buffer)
Capabilities	none — sending is `outbuf`'s job

Assembles the ANSI control bytes to send to a terminal, and computes how much the screen changed and how many cells characters occupy. Used in TUIs to redraw only what changed instead of everything (chapter 37).

> **This module writes nothing to the screen**
>
> It only builds bytes and puts them into the caller's buffer. Every op is `effects none`. Actually sending to the screen is done by the side holding `cap io` (`outbuf`) — assembly and output are separate. Missing this leads to the dead end "the code runs but nothing appears". In return, all escape assembly and diffing is verified by byte comparison without a screen. Key input and raw mode are `tty`.

```
use term as t .

let p option u64 . be t.goto buf 0 2 4 .
guard is_some p . else return 1 .
```

Convention. Sequence ops follow `fmt`'s convention — `(buf, pos, …) → option u64` (new pos). Passing one op's return as the next op's `pos` chains sequences, and the last pos is the finished sequence's length. **All or nothing** — short of room, not a single byte is written. A half-written escape is silently wrong output: the terminal reads the rest as characters and debris like `[3;5H` appears. Coordinates are taken **0-based** and emitted in ANSI's 1-based form — 1-based is the wire's business.

op	Sequence	Size
`clear buf pos`	`ESC[2J`	4
`goto buf pos row col`	`ESC[<row+1>;<col+1>H`	4 + decimal digits of both numbers
`sgr buf pos n`	`ESC[<n>m` — 0 reset · 1 bold · 4 underline · 7 reverse	3 + digits of n
`color256 buf pos n bg`	`ESC[38;5;<n>m` (foreground) · `ESC[48;5;<n>m` (background)	8 + digits of n
`cursor buf pos show`	`ESC[?25h` · `ESC[?25l`	6

Table 50.1 — Sequence ops and the bytes they need

Screen diff. `diff prev nxt w out pos` compares the previous screen `prev` with the new screen `nxt` (same length; one-byte cells in rows of width `w` laid end to end) and, for each **run of consecutive changed cells within a row**, writes `goto` + the changed bytes. Runs never cross row boundaries. No change gives 0 bytes — **0 bytes means "leave the screen alone"**, a normal answer. Conditions: `len prev == len nxt`, `w > 0`, `len prev % w == 0`. UTF-8 screens use `diff_row_utf8 prev nxt row out pos` per row — emitting only changed **code point runs** as `goto(row, code point column)` + those `nxt` bytes. Wider cells change what a diff means, so rather than altering `diff` it sits beside it as a separate op — code relying on the old contract must not silently change.

op	Answer	Failure
diff prev nxt w out pos	some <new pos>	condition violated · out short = none, **out contents undefined**
diff_row_utf8 prev nxt row out pos	some <new pos>	invalid UTF-8 · out short = none, **out undefined**
row_cells row	number of code points	invalid UTF-8 = none
col_off row c	byte offset of the c-th code point	invalid UTF-8 · row shorter than c cells = none
cp_width cp	0 · 1 · 2 cells	no failure — 1 if unknown
row_width row	total cells of a row	invalid UTF-8 = none
fit_width row cols	**bytes** fitting within cols cells (never cutting a character)	invalid UTF-8 = none · some 0 if not even one cell fits
cluster_len s at	byte length of the character cluster starting at at (at least 1)	out of range · not a boundary · invalid UTF-8 = none
row_clusters row	number of character clusters in a row	invalid UTF-8 = none

Table 50.2 — Diff and counting ops — all effects none

Confusing these four puts the cursor in the wrong place. Bytes ≠ code points (row_cells) ≠ cells (row_width) ≠ characters (row_clusters). "한글" is 6 bytes, 2 code points, **4 cells** and 2 characters; "e" + U+0301 is 3 bytes, 2 code points, **1 cell** and **1 character**. cp_width is 0 for combining marks and format characters (is_zerowidth of unicode), 2 for Korean, CJK and fullwidth symbols, and 1 otherwise. Even a wrong width guess of 1 is a usable default, so it never fails — unlike property tests. A character cluster is one leading code point plus all following zero-width code points — backspace and cursor movement must use this unit, or only the accent gets deleted or the cursor stops inside a character.

```
proc diff_frame input out mut slice u8 . . output u64 . effects none . do
  guard ge (len out) 32 . else return 90 .
  let prev slice u8 be "aaaaaaaaaa" .
  let nxt  slice u8 be "aaaaaaxyaa" .
  let p option u64 . be t.diff prev nxt 5 out 0 .
  guard is_some p . else return 1 .
  guard eq (some_value p) 8 . else return 2 .
  let d option u64 . be t.diff_row_utf8 "가나다" "가라다" 0 out 0 .
  guard is_some d . else return 3 .
  guard eq (some_value d) 9 . else return 4 .
  return 42 .
end
```

The first diff changes only columns 1 and 2 of the second row: ESC[2;2H (6) + xy (2) = 8 bytes; the UTF-8 row diff is ESC[1;2H (6) + 라 (3) = 9 bytes. A render loop computes nxt → diff → on some p sends subslice out 0 p through outbuf and swaps prev and nxt. Forgetting the swap redraws the same runs every frame.

Counter-example. Sending `out` when `diff` returned `none`

Only the two diffs are not all-or-nothing — if `out` runs short midway, written bytes remain. Sending that buffer after `none` pushes out a half-written escape and breaks the screen. Retry with a larger buffer, or `clear` and redraw everything.

Counter-example. Swapping `prev` and `nxt`

Same type, same length, so neither compiler nor runtime catches it, and with equal byte counts tests may pass. The only symptom is the screen reverting to old content. Memorise the order `diff <previous> <new>`.

Counter-example. Cutting by bytes or counting cells by code points

`subslice row 0 3` has no guarantee that 3 bytes is a character boundary, sending half a character — `fit_width` keeps boundaries. `row_cells "한글"` is 2 but the cells are 4, so aligning by code points is off by two — ask `row_width` for cells.

Cautions. If only `none` keeps coming, suspect the buffer size — measure by **room remaining after `pos`**. Estimate `out` at worst as rows × maximum goto width + screen size. Mistaking the no-change `some pos` for failure and falling back to a full redraw keeps the screen right but flickers every frame. Do not pass coordinates already 1-based — everything shifts by one cell. Feeding a UTF-8 screen to the one-byte-cell `diff` can cut multibyte characters — the two diffs are separate ops with different contracts. **Not built** — the remaining cluster rules such as flags (regional indicator pairs), emoji ZWJ sequences and Hangul jamo composition, and diffs of attribute (colour) cells. Better to write down what is not done than to pretend a partial rule is complete.

`sortlib` — in-place quicksort

Source	`lib/sort.low`
Layer	L0 — pure computation
Capabilities	none

Sorts a `u64` slice **in place** in ascending order. Used before binary search (`searchlib`) and to present results in order. The one public op is `sort`.

```
use sortlib .

sortlib.sort s .
```

> **Maturity `standard` — the repository's first `standard`**
>
> The current surface (just `sort`) is **kept**. Changing names, arguments or meaning needs a separate compatibility decision. That does not mean it will not grow — a comparator and other element types do not exist yet, and would arrive **by addition**. And **performance is not promised** — today it is quicksort. Other element types are `sortgen`'s job.

Sorting is **pure computation** almost every tool needs, and it reaches nothing in the world. So it is a library, not a builtin — if it can be written in Lowent, it is a library. It was built with zero new builtins.

op	Shape	Failure
`sort`	`proc (s mut slice u64) → void`, `effects none` — in-place ascending	none (empty and one-element slices do nothing)
`qsort` (internal)	`proc (s mut slice u64, lo0 u64, hi0 u64) → void`	not exported — calling from outside is `E-VISIBILITY`

Table 50.1 — Ops of `sortlib`

Depth is bounded by O(log n) — the point of this implementation. Naive Lomuto partitioning on already sorted input always picks an end pivot, so recursion gets O(n) deep and can exceed the native depth limit. So it **recurses on the smaller side and loops on the larger**. Only halves or smaller pile up in recursion, so it cannot exceed log n. This is for resource safety, not correctness — it may slow down, but it does not blow up. The internal `qsort` handles the half-open range `[lo0, hi0)` and does nothing if `hi0 > len s` or `lo0 >= hi0` — even a reversed range never touches out of bounds, so VM and native give the same answer (nothing).

Why no new array is returned. That would need allocation, which this layer does not do. The same is why no comparator is taken — if another order is needed, transform the values beforehand.

A self-verifying pattern — is it ascending **and** is the sum preserved. It catches order defects and element loss or duplication together.

```
proc sort_verify input s mut slice u64 . . output u64 . effects none . do
  var pre u64 be 0 .
  var i u64 be 0 .
  while lt i (len s) . do
    set pre (wrap_add pre (index s i)) .
    set i (add i 1) .
```

```
  end
  sortlib.sort s .
  var post u64 be 0 .
  var sorted u64 be 1 .
  var j u64 be 0 .
  while lt j (len s) . do
    set post (wrap_add post (index s j)) .
    if gt j 0 . do
      if gt (index s (sub j 1)) (index s j) . do set sorted 0 . end
    end
    set j (add j 1) .
  end
  if ne pre post . do set sorted 0 . end
  return sorted .
end
```

Counter-example. Sorting a subrange with the internal op

`sortlib.qsort s 0 (len s)` is `E-VISIBILITY`. Cut a subrange with `subslice` and give it to `sort` — `sortlib.sort (subslice s 2 7) .`

Counter-example. Passing an immutable slice or expecting a return value

Passing `input s slice u64` is a type error — in-place sorting needs `mut slice u64`. If the original must survive, sort a copy. `let r … be sortlib.sort s` is a compile error too — it is `output void`, and the result is `s` itself.

Cautions. The input is destroyed. It is not stable (the original order of equal values is not guaranteed — unobservable with `u64` today, but that changes once key extraction exists). Time is O(n log n) on average and O(n^2) worst case; extra memory is only O(log n) recursion frames. **What is checked** — sortedness and permutation (tests), VM/native agreement on arbitrary inputs (differential testing).

`sortgen` — generic sorting (the type brings the comparison)

Source	`lib/sortgen.low`
Layer	L0 — pure computation
Capabilities	none

Sorts even when elements are not `u64`. What is smaller **the type says for itself**. Used to order records by some field — for plain `u64` elements `sortlib` is simpler and faster. Three things go together — ① declare that the type satisfies `ordered`, ② write that type's `less`, ③ call with the type as an argument.

```
use sortgen .

struct keyed do
  satisfies sortgen.ordered .
  k u64 .
end

fn keyed.less input a keyed . input b keyed . output bool . do
  return lt (field a k) (field b k) .
end

proc sorted3 input s mut slice keyed . . output u64 . effects none . do
  guard ge (len s) 3 . else return 90 .
  sortgen.sort_by keyed s .
  let f0 keyed be index s 0 .
  let f1 keyed be index s 1 .
  let f2 keyed be index s 2 .
  return add (mul 100 (field f0 k)) (add (mul 10 (field f1 k)) (field f2 k)) .
end
```

Why the comparison is not passed as a function value. The common answer (C's `qsort`, C++ comparators) passes a comparison function as a value. This language does not. It has no first-class functions, and even with them there would be an **indirect call** — free-looking at the call site, while the cost of going through a function pointer hides. This language's answer is **comptime type parameters** + **trait bounds** (chapter 22, chapter 23). Each call creates an op specialised for that type, and the comparison is embedded as a direct call. Zero indirect calls, zero space for a comparator.

op	Shape	Failure
`ordered`	trait — `less (self, self) → bool`	unsatisfied = `E-BOUND-UNSAT` (compile time)
`sort_by`	`(comptime t, mut slice t) → void`, `requires ordered t` — insertion sort (stable)	none
`sort_fast`	same shape — quicksort, for large arrays	none
`lower_by`	`(comptime t, slice t, key t) → u64`	none — the insertion point if absent (may be `len s`)
`find_by`	`(comptime t, slice t, key t) → option u64`	`none` if absent

Table 50.1 — Ops of `sortgen`

If `less a b` is true, a comes before b. The relation must be a **strict weak order** — in particular `less a a` must be false. Without `requires ordered t` the body would call `less` without anyone asking whether it exists. With the bound, calling with a type lacking it is rejected at compile time.

The story of `sort_fast`. At first a generic quicksort was not built on the belief that partitioning needs a **copy** of the pivot. Wrong — hold the pivot by **position** (index) and re-read it at each comparison. During the partition loop the pivot stays at `hi - 1`, and all moving is done by `swap`.

> **Counter-example. Forgetting `satisfies` or passing `u64` directly**
>
> Calling `sort_by` with a struct lacking `satisfies sortgen.ordered .` or with `u64` is `E-BOUND-UNSAT`. `u64` has nowhere to attach a method, so it cannot satisfy a trait — wrap it in a one-field struct (layout stays 8 bytes), or use `sortlib.sort`.

> **Counter-example. `less` returning true for equal values**
>
> `return le (field a key) (field b key) .` compiles, but with two or more equal values the loop may never end. The comparison must be `lt`.

Cautions. Descending order and multiple keys are written in `less` — adding a mode argument is entropy itself. One copy of code is produced per type (the price of monomorphisation, in exchange for no indirect calls). **Reading gives a view** — `let a t be index s j` is a window onto that position, not a copy, so elements are swapped with `swap` (two `set`s would overwrite themselves). Take care if you write a sort yourself — after one swap the same names `a` and `b` see different values. Decide once per step. Comparing twice makes the second comparison see the already swapped values, and an element moves down one place and stops — a partial sort that, with three elements and a single key, happens to give the right answer and hides the defect (it really did hide, until a multi-key example revealed it). It is separate from `sortlib` because bringing in generic templates would have excluded every file using `sortlib` from some comparison checks.

`searchlib` — binary search on sorted slices

Source	`lib/search.low`
Layer	L0 — pure computation
Capabilities	none

Finds values in a **sorted** slice by binary search. Probe the middle, decide whether the target is left or right, and search only the remaining half — about twenty steps even for a million elements. In exchange, elements must already be in line to choose a direction. It is the partner of `sortlib`, and together they make **sorted sets and maps** — unlike a hash map, order is kept, so range queries and "the next larger value" work. The file is `search.low`, but the module name is `searchlib`.

```
use searchlib .
use sortlib .

sortlib.sort s .
let i option u64 . be searchlib.bsearch s 42 .
```

> **Sortedness is not checked**
>
> The premise that `s` is ascending is **the caller's job**. If broken, answers are wrong but nothing stops — no out-of-range access happens, so there is nothing to stop for; instead **a silently wrong answer** comes out. "`none` for a value that is definitely there" almost always means it was not sorted.

op	Shape	Answer
bsearch	`(s slice u64, target u64) → option u64`	index of `target`, or `none`. With duplicates, **one** of them (which one is unspecified)
lower_bound	`(s slice u64, target u64) → u64`	first index at or above `target`. `len s` if all are smaller

Table 50.1 — Ops of `searchlib` — all `fn · effects none`

How to choose — "is it there, and where" is `bsearch`; "where would it go" or "where do values at or above this start" is `lower_bound`. `lower_bound` is both **insertion point** and **start of a range** — `[lower_bound s a, lower_bound s b)` is the index range of elements with `a ≤ x < b`. It is not an `option` because the question has no "not found" (the end index is an answer too). There is no upper_bound, insert or delete — one lower bound gives membership, insertion point and range queries.

The implementation runs on the half-open range `[lo, hi)` with midpoint `lo + (hi - lo)/2` — `(lo + hi)/2` can overflow at large indices (the classic defect).

```
fn count_in input s slice u64 . input a u64 . input b u64 . output u64 . do
  let i u64 be searchlib.lower_bound s a .
  let j u64 be searchlib.lower_bound s b .
  guard lt i j . else return 0 .
  return sub j i .
end
```

Counter-example. Not separating `bsearch`'s `none`

`return some_value r .` compiles, and stops with `E-VM-NONE` the moment nothing is found. Testing only with present values never triggers it — seeing an `option`, add `guard is_some` (chapter 11).

Counter-example. Giving meaning to `bsearch`'s index among duplicates

In `[5,5,5]`, `bsearch s 5` may yield 0, 1 or 2. If you need "the position of the first 5", use `lower_bound s 5`.

Counter-example. Indexing directly with `lower_bound`'s result

If not found, `i` is `len s`, a position that does not exist — `index s i` stops with `E-VM-BOUNDS`. It only fires when the value exceeds the table's maximum, so it tends to hit with real data. Put `guard lt i (len s) .` first.

Cautions. It cannot be used as is on descending data (there is no place to swap in a comparator). **Sort once, search many times** — calling `sort` before each search loses the whole gain. After sorting, changing one element with `set` requires sorting again. Keep the order when subtracting two lower bounds — `sub` underflows. As pure ops, the same slice can be searched from many places at once (while nobody writes). Other element types are `lower_by` and `find_by` of `sortgen`.

`hashmap` — `u64` → `u64` hash map

Source	`lib/hashmap.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

A hash map storing and finding `u64` values by `u64` keys. Used to look things up quickly by number (id → count, handle → state). String keys are `strmap` the container that chooses element type and allocator is `mapgen` (chapter 34).

```
let stored bool be hashmap.put b 7 42 .
let v option u64 . be hashmap.lookup b 7 .
```

Two things to know

① The map makes no memory of its own — the caller provides storage `b` and passes it every time. ② A new map **must start all zeros**. Break either and answers go wrong without errors — a slice full of garbage reads as "a map that already holds things", so keys never inserted are found, or a key just inserted comes back `none`.

Open addressing. No linked list per key; if a slot is taken, insertion moves to the next slot of the same table (linear probing) — one buffer suffices, matching the caller-owned discipline. The backing is one `mut slice u64` laid out as `[key0+1, val0, key1+1, val1, …]` (N slots = `len / 2`). Key slots store `k+1`, so `0` marks an empty slot and **key 0 can be stored**. Deletion overwrites the key slot with a **tombstone**, `MAX_U64` — turning it back into an empty slot would break the trail of keys pushed further along by collisions, making later keys unfindable. `lookup` skips tombstones, `put` reuses them, traversal does not count them. The hash is multiplicative (golden ratio constant).

op	Shape	Failure
`put`	`proc (b mut slice u64, k u64, v u64) → bool`	full · reserved key → `false`
`lookup`	`proc (b slice u64, k u64) → option u64`	absent · reserved key → `none`
`del`	`proc (b mut slice u64, k u64) → bool`	absent · reserved key → `false`
`size`	`fn (b slice u64) → u64`	slot count (not the item count)
`occupied_at`	`fn (b slice u64, slot u64) → bool`	out of range → `false`
`key_at` · `val_at`	`fn (b slice u64, slot u64) → u64`	meaningful only when occupied
`rehash`	`proc (nb mut slice u64, ob slice u64) → bool`	`false` midway if `nb` is too small

Table 50.1 — Ops of `hashmap` — all `effects none`

- `put` updates in place for an existing key. It remembers the first tombstone met while probing and inserts there once the key is confirmed absent further on — tombstone reuse keeps clusters from growing.
- **It does not grow by itself.** When full, `put` returns `false` (no stop). To grow, the caller prepares a larger zeroed backing and calls `rehash nb ob`. Only live items move, so tombstones disappear too (compaction). The old backing is only read, so it stays intact on failure.
- **Two keys cannot be stored.** `MAX_U64` and `MAX_U64 - 1` are reserved by the encoding — the price of getting deletion without a separate status byte.
- Traversal scans `0 .. size` and handles only `occupied_at` slots. Order is hash order — for sorted enumeration, extract the keys and sort them with `sortlib`.

```
proc hm_sum input b mut slice u64 . . output u64 . effects none . do
  var i u64 be 0 .
  while lt i (len b) . do
    set (index b i) 0 .
    set i (add i 1) .
  end
  guard ge (div (len b) 2) 4 . else return 0 .
  guard eq (hashmap.put b 10 100) true . else return 0 .
  guard eq (hashmap.put b 20 222) true . else return 0 .
  guard eq (hashmap.del b 10) true . else return 0 .
  var slot u64 be 0 .
  var total u64 be 0 .
  while lt slot (hashmap.size b) . do
    if hashmap.occupied_at b slot . do
      set total (add total (hashmap.val_at b slot)) .
    end
    set slot (add slot 1) .
  end
  return total .
end
```

> **Counter-example. Reading slots without `occupied_at`**
>
> `key_at` and `val_at` of empty or tombstone slots are meaningless (an empty slot's `key_at` goes as far as `0 - 1`), so totals go wrong without errors. Always filter traversal with `occupied_at`.

> **Counter-example. Ignoring `put`'s `false`**
>
> A full map answers `false` instead of stopping, so the line passes silently; later `lookup` returns `none` and an unchecked `some_value` stops with `E-VM-NONE`. The stop is far from the cause — `guard eq (hashmap.put …) true .` should be habit.

Cautions. Why `lookup`, not `get`, and `size`, not `capacity` — `get` is the payload enum destructuring word and `capacity` is syntax in `alloc_bytes … capacity n`, so ops cannot have those names. Repeated insert and delete pile up tombstones and lengthen probes — move to a same-size backing with `rehash` to compact. With an odd backing length the last u64 is unused.

`strmap` — string-keyed hash map

Source	`lib/strmap.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

A hash map storing and finding `u64` values by byte string (string) keys — word counts, symbol tables, configuration maps. It widens `hashmap` to variable-length byte string keys. Key bytes are **copied** into an arena on insertion, so they are not tied to the original string's lifetime.

```
let stored bool be strmap.put slots keys "apple" 1 .
let v option u64 . be strmap.lookup slots keys "apple" .
```

The caller prepares two buffers. `slots` must start all zeros.

- `slots` (`mut slice u64`) — the slot table. `slots[0]` is **the arena cursor** (how many key bytes are used); real slots start at index 1, one slot = three u64 `[keyoff, keylen, value]`. N slots = `(len slots - 1) / 3`.
- `keys` (`mut slice u8`) — the key byte arena. Bytes of many keys laid end to end in one buffer (a bump that only moves the cursor forward).

`keylen` has three meanings — `0` = empty, `MAX_U64` = tombstone, anything else = the length of a live key. A real key length cannot be MAX, so it serves as the tombstone value without reserving anything. The hash is FNV-1a 64 with linear probing.

op	Shape	Failure
put	`proc (slots mut slice u64, keys mut slice u8, k slice u8, v u64) → bool`	full · arena short · empty key → `false`
lookup	`proc (slots slice u64, keys slice u8, k slice u8) → option u64`	absent · empty key → `none`
del	`proc (slots mut slice u64, keys slice u8, k slice u8) → bool`	absent · empty key → `false`
size	`fn (slots slice u64) → u64`	`len slots < 4` → 0
occupied_at	`fn (slots, slot u64) → bool`	out of range → `false`
keylen_at · keyoff_at · val_at	`fn (slots, slot u64) → u64`	out of range → 0
rehash	`proc (ns mut slice u64, na mut slice u8, os slice u64, oa slice u8) → bool`	`false` midway if `ns` or `na` is too small

Table 50.1 — Ops of `strmap` — all `effects none`

- `put` updates only the value for the same key (length and bytes equal) — no new slot, no arena use. New keys are copied into the arena. The first tombstone met while probing is reused.
- `del` overwrites keylen with a tombstone and zeroes the value. **The old key bytes stay in the arena** — reclamation happens when `rehash` moves to a new arena (compaction).
- Read key bytes with `subslice keys off (add off klen)`. That is a view into the arena, valid only while the arena lives.

Counter-example. An empty string key

`strmap.put slots keys "" 7` is always `false`. `keylen 0` marks an empty slot, so an empty key cannot exist in the encoding. `lookup ""` is always `none` too.

Counter-example. Traversing by `keylen > 0` after deletion

A tombstone's keylen is MAX, so `gt keylen 0` is true — deleted items creep into totals. Filter with `occupied_at` in any map that has ever used `del`.

Counter-example. Touching `slots[0]` directly

Tampering with the arena cursor makes the next `put` copy key bytes to the wrong place. Nothing stops; it is silently wrong. Initialise as "all zeros", then modify only through ops.

Cautions. The arena is a bump — repeatedly inserting and deleting different keys can make `put` return `false` for lack of arena even with free slots. One `false` from `put` does not distinguish slot shortage from arena shortage. Estimate sizes as `3N + 1` u64 for N slots and, for the arena, the sum of live key lengths plus dead bytes. A real use reads a file, splits words with `strings` and counts frequencies (chapter 34).

vecs — growable byte vectors with caller-held buffers

Source	`lib/vec.low`
Layer	L1 — allocator via `using`
Capabilities	none — an allocator is handed in

A byte vector that, when short of room, **takes a larger place and moves**. Used to gather byte strings of unknown final size. What is missing is reallocation, not growth — take a larger place and move the old contents, which can be written in Lowent. So it is a library, not a builtin, and the growth policy (doubling) belongs to the library too.

	vecs	vecgen
Storage	**held by the caller** — buffers are passed to ops	owned by the container
Growth	the new place is **returned as a value** (the caller rebinds)	swapped inside
Allocator	handed over with `using` on each call	held as a field

Table 50.1 — `vecs` and `vecgen` — ownership is the difference

`vecs` is for when you want to hold buffer lifetimes **directly** inside a region. A vector lives as **a pair of two values**: `vec_u8` (a single length `len`) and `mut slice u8` (storage). Capacity is not stored separately — `len buf` is the capacity. Push maintains the invariant `v.len ≤ len buf`.

op	Shape	Failure
vec_u8	struct — `len u64`	—
next_cap	`fn (cur u64) → u64` — 8 from 0, otherwise double	none
push_byte	`proc (comptime a, using al a, v mut vec_u8, buf mut slice u8, x u8) → option (mut slice u8), effects state via a, requires allocs.byte_allocator a`	`none` on OOM — `v` and `buf` unchanged

Table 50.2 — Ops of `vecs`

With room, `push_byte` writes in place and returns `some buf`; when full it takes double with `send al reserve (next_cap (len buf))`, copies the old contents and returns `some <new buffer>`. The allocator is handed over with a `using` clause, not a positional argument — the type `a` is filled from that source and monomorphised, so passing it costs nothing (chapter 35). No hidden allocation, so code without an allocator cannot call this op.

```
module vecgrow .
use vecs .
use allocs .

proc main input al cap allocator . input out cap io . output u8 . effects io alloc state . do
  let memopt option mut slice u8 . be alloc_bytes al capacity 256 .
  guard is_some memopt . else return 1 .
  let mem mut slice u8 . be some_value memopt .
  var bump allocs.bump_bytes be spawn actor allocs.bump_bytes . .
  var c u64 be send bump init mem .
  let b0 option mut slice u8 . be send bump reserve 2 .
  guard is_some b0 . else return 2 .
  var buf mut slice u8 . be some_value b0 .
  var v vecs.vec_u8 be make vecs.vec_u8 do len 0 . end .
```

```
  var i u64 be 0 .
  while lt i 10 . do
    let r option mut slice u8 . using bump be vecs.push_byte v buf (narrow u8 (add 65 i)) .
    guard is_some r . else return 3 .
    set buf (some_value r) .
    set i (add i 1) .
  end
  let m u64 be write_out out 1 (subslice buf 0 (field v len)) .
  return narrow u8 (field v len) .
end
```

Starting at 2 bytes and pushing 10, it grows three times, $2 \to 4 \to 8 \to 16$, and prints `ABCDEFGHIJ`.

> **Counter-example. Not rebinding to the returned buffer**
>
> Forgetting `set buf (some_value r) .` compiles and runs, but the result is silently wrong. After growth, passing the old `buf` keeps `v.len` counting against the new buffer while writes go to the old (small) one. **The return value is the next buffer** — the most dangerous mistake in this module.

> **Counter-example. Ignoring OOM · calling from a pure fn**
>
> Unwrapping `some_value r` unchecked stops with `E-VM-NONE` when the allocator runs dry. Calling from an `effects none fn` is `E-EFFECT-CALC` — `push_byte` is a proc with `effects state`. Passing a source that does not satisfy the trait (such as `u64`) is `E-BOUND-UNSAT`.

Cautions. Growth is copying — one push is O(len) worst case, but doubling gives amortised O(1). If the size is known ahead, starting that large is cheaper. **Old buffers are not reclaimed** — growing to n on a bump consumes about 2n of backing, and old buffers are swept away together when the `region` ends (chapter 18). Header and buffer are a pair — nothing prevents mixing those of different vectors. There is no element type generalisation, pop, insert, remove or shrinking.

spsc — lock-free SPSC ring buffer

Source	`lib/spsc.low`
Layer	L0 — pure computation (the caller's storage)
Capabilities	`cap atomic`

The simplest way to pass values between threads is a lock. But with exactly one producer and one consumer no lock is needed — the producer writes only `tail`, the consumer only `head`, and each only **reads** the other's atomically. That is SPSC (single-producer single-consumer), and almost the only lock-free structure that is honestly short.

> **One producer and one consumer is the contract**
>
> If two threads `push` or two threads `pop` it is no longer SPSC, and this module cannot prevent it. That is why convenience ops like `peek` or `drain` were not added — the more convenience ops, the more ways to break that contract. In a standard library a small surface is not laziness but a means of keeping the contract. There is no MPSC, MPMC or seqlock — with CAS retries and ABA attached, putting them in the standard would make unverified code standard.

One slot is always left empty. So the real capacity is `len buf - 1`. In return `head == tail` **always** means "empty" and is never confused with full.

op	What it does	Returns
`spsc_capacity buf`	Maximum number it can hold	`len buf - 1`
`spsc_push k ctl buf v`	Push (**producer only**)	1 success · 0 full
`spsc_pop k ctl buf out`	Pop (**consumer only**), value in `out[0]`	1 success · 0 empty
`spsc_count_about k ctl buf`	Current count (an observation)	count

Table 50.1 — Ops of `spsc`

`ctl` is a `mut slice u64` of length at least 2, with `[0]` = head (consumer) and `[1]` = tail (producer). `k` is `cap atomic` — the right to touch shared places atomically. Why `count_about` says `about` — it may already have changed by the time it is read. If the name does not say so, users trust the value.

What holds this code up. Two proofs. First, wrapping an index with `mod … (len buf)` always lands in range — `lowentc --emit-proof lib/spsc.low` records that the checks on both ring buffer accesses (`index`, `index.store`) and on the two index-advancing additions were removed by proof (chapter 41). Second, write data → publish index → observe index → read data carries the data under the weak memory model (RC11) — and the correctness of the algorithm itself (on success the resource moves into the buffer, on failure it comes back) borrows an existing proof (gpfsl's `circ_buff`). The correspondence that our `spsc_push` matches that proof's code line by line was checked by a person reading it, not by machine (chapter 47).

Orderings are exactly as strong as needed. When the producer reads head it uses `acquire` (published by the consumer), when it reads its own tail `relaxed` (a value I wrote), when it publishes tail `release` (the preceding data write must be visible first). The consumer side is symmetric. Only two of the four places need synchronisation. This notation (`order <name>`) was once unreachable because of a parser defect, found and fixed while trying to use this library — a feature nobody uses might as well not exist.

What is checked — pop from empty fails, push to full fails, FIFO order, and correctness after wrapping around the ring. Those tests are single-threaded (the VM is single-threaded). Correctness of memory ordering with two real threads comes from proof, not from running — the two are not mixed.

hash — hashing (hash map slots · corruption detection · SHA-256)

Source	`lib/hash.low`
Layer	L0 — pure computation
Capabilities	none

Produces numbers from byte strings. Three things of different character live in one module, and mixing them goes wrong in the dangerous direction.

> **`of` and `crc` are not cryptographic**
>
> `of` (FNV-1a 64) is for **hash map slots** — fast and well spread, but not cryptographic. `crc` (reflected CRC-32) detects **accidental corruption** — matching a desired CRC is easy, so it does not stop forgery. `digest` and `digest_ok` (SHA-256) are the cryptographic hash; proof of integrity lives there. Keyed authentication is not `digest` either but `hmac` — a hash alone cannot answer **who sent it**.

```
use hash .

fn slot input key slice u8 . input nslots u64 . output u64 . do
  return hash.bucket_of key nslots .
end
```

Why the algorithms are fixed. A hash looks right whenever a plausible number comes out. If not fixed, nothing is left to check. Because they are fixed, standard check values become the tests' reference (CRC-32 of `"123456789"` → `0xCBF43926`), and if VM and native diverge it shows on the spot.

op	Shape	Use
`of`	`(data slice u8) → u64`	FNV-1a 64 — hash map slots
`bucket_of`	`(data, nslots u64) → u64`	Choosing a slot (remainder)
`bucket_mask`	`(data, mask u64) → u64`	When the slot count is a power of two
`crc`	`(data slice u8) → u64`	Reflected CRC-32 — corruption detection
`crc_ok`	`(data, want u64) → bool`	Compare with a check value
`maybe_same`	`(a, b slice u8) → bool`	Are the hashes equal — not proof of equality
`digest`	`(data, out mut slice u8) → u64`	SHA-256 — bytes written (32)
`digest_ok`	`(data, out mut slice u8, want slice u8) → bool`	Compare with a received digest

Table 50.1 — Ops of `hash`

Why `digest_ok` exists — the right use of a hash is not a round trip but **comparison**. Recomputing and matching against what was received is the only correct use, so that shape is an op (if `want` is not 32 bytes it is simply `false` — the contract starts with the length). Why `maybe_same` says `maybe` — equal hashes do not mean equal values. Calling it `same` would have made the name a lie.

Not built — keyed hashing (that is `hmac`), seedable hashing (HashDoS defence), streaming (feeding in pieces), cryptographic hashes other than SHA-256.

`math` — floating-point maths

Source	`lib/math.low`
Layer	L0 — pure computation (host only)
Capabilities	none

Adds commonly used things on top of builtins like `sqrt`, `sin` and `exp` — comparison (`close`), constants (`pi`, `e`), angle conversion, hypotenuse, logarithms with a base, and linear interpolation. It has no capabilities but links to the C maths library, so it is **host only** — targets without an operating system may have no floating point at all (`cortex_m` is `no_float`).

> **Do not compare with `==` — use `close`**
>
> ```
> guard math.close (math.hyp 3.0 4.0) 5.0 0.000001 . else return 1 .
> ```
>
> "Equal" is not well defined for floating point (chapter 4). So this module provides `close(a, b, tol)`, and the library's own tests judge with it.

Why checking is not bit-exact. C maths libraries **may differ in the last digits between implementations**. Claiming exact bits would cause false failures on other machines and libcs, with tests failing while the tool is fine. So this module is checked against **known answers** (`sin(0) = 0`) and **identities** (`sin²+cos² = 1`, `exp(log x) = x`), judged with `close`. How close counts as equal (`tol`) is the caller's question, not something the library should decide.

op	What it does
`close`	`|a - b| ≤ tol` — this module's way of comparing
`pi` · `e`	Constants
`deg_to_rad` · `rad_to_deg`	Angle conversion
`hyp`	`sqrt(x² + y²)`
`log_base`	Logarithm with a base (`log x / log b`)
`lerp`	Linear interpolation `a + (b - a)·t`

Table 50.1 — Ops of `math`

Used directly as builtins — `sqrt`, `abs`, `floor`, `ceil`, `round`, `sin`, `cos`, `exp`, `log`, `pow`.

Not built — `atan2`, `asin`, `acos`, `tan`, `log2`, `log10`, `cbrt`, complex numbers, fixed point, an `f32`-only face. If needed they attach as one builtin plus one line here. `hyp` computes `sqrt(x²+y²)` as is — unlike libm's `hypot` it does not rescale to avoid overflow, so it can answer differently for very large values.

`random` — random numbers (reproducible sequences · OS entropy)

Source	`lib/random.low`
Layer	L0 — pure computation · OS entropy via capability
Capabilities	`cap random` for `bytes` · `seed_from_os`

> **Do not mix the two — this warning is the whole module**
>
> `step`, `below_biased` and `coin` are a reproducible sequence made with **splitmix64**. They need no capability and serve tests, simulations and shuffles. `bytes` and `seed_from_os` are **OS entropy**. They need `cap random` and serve keys, nonces and tokens. Tests must be reproducible and keys must not be predictable — one word cannot do both, so the names are split. Use a value made by `step` as a key and anyone who knows the seed knows that key, and seeds usually remain in code or logs.

The reproducible side. Takes a state (= seed) and returns the next state. The caller carries the state, so the same seed always gives the same sequence.

```
var s u64 be 12345 .
set s (random.step s) .
let c bool be random.coin s .
```

The unpredictable side. Callable only with `cap random`. It answers the number of bytes filled, and **0 if it could not fill** — taking 0 and using the buffer anyway means using an uninitialised buffer as a key.

```
proc make_key input k cap random . input key mut slice u8 . output bool . effects none . do
  return eq (random.bytes k key) (len key) .
end
```

op	Shape	Notes
`step`	`(seed u64) → u64`	splitmix64 — algorithm fixed (so check values can be the reference)
`below_biased`	`(seed u64, bound u64) → u64`	The name **confesses the bias** — not this if you need uniformity
`coin`	`(seed u64) → bool`	Heads · tails
`bytes`	`(cap random, dst mut slice u8) → u64`	Bytes filled. **0** = **failure**
`seed_from_os`	`(cap random, scratch mut slice u8) → u64`	One seed from the OS — makes the starting point of a reproducible sequence unpredictable. `scratch ≥ 8`

Table 50.1 — Ops of `random`

Why `below_biased` says `biased` — narrowing a range with the remainder makes small values come up slightly more often. Rather than hide it, the name says it.

Tested by summoning failure. The environment variable `LOW_HOST_FAULT="random:err"` makes entropy unavailable (→ 0), and `LOW_HOST_FAULT="random:short=3"` fills only 3 bytes. **Partial filling is the quietly frightening case** — a key made with half the buffer still holding old values. So checking the count filled is the contract (chapter 28).

Not built — uniform range sampling (rejection sampling), shuffles, distributions (normal and so on), serialising reproducible sequences, a cryptographic CSPRNG (`bytes` asks the OS; this module does not make it).

`hmac` — HMAC-SHA256 and HKDF

Source	`lib/hmac.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

It does two things. **HMAC** is a token only someone knowing the key can make (`mac`), answering "who sent it" (RFC 4231). **HKDF** is the standard way to split several keys out of one secret (`extract` + `expand1`) (RFC 5869).

> **What it promises and what it does not**
>
> It does not promise constant time and has not been audited. **Comparing MACs is the caller's job** — compare without early return (accumulating bytes with XOR). A comparison returning at the first differing byte lets the tag be learned one byte at a time. `expand1` produces **only one block (at most 32 bytes)**.

Why a hash will not do. `digest` (SHA-256) of `hash` answers only "is the content unchanged". Anyone can recompute it, so it cannot answer who sent it. A key must be involved to open that question. The caller holds the backing (`scratch`). The size requirement is enforced by **guards**, not documentation, so if short it answers 0 and writes nothing.

op	What it does	Requires
`mac`	`HMAC-SHA256(key, msg)` → 32 bytes into `out`	`out ≥ 32` · `scratch ≥ 64 + len(msg)`
`key_block`	Makes K′ (the key fitted to 64 bytes)	`pad ≥ 64`. Keys longer than 64 are hashed down
`extract`	HKDF-Extract: `PRK = HMAC(salt, ikm)`	same as `mac`
`expand1`	First block of HKDF-Expand: `T(1) = HMAC(prk, info ‖ 0x01)`	`scratch ≥ 130 + len(info)` · `len(info) ≤ 60`

Table 50.1 — Ops of `hmac`

Why `expand1` is one block only. TLS 1.3's `HKDF-Expand-Label` uses only 32 bytes or fewer (keys, IVs and finished values alike). So the loop chaining T(2) was not built. **Why cut at 60.** `msg` lives inside `scratch` (from 128) and `mac` overwrites `scratch[0 .. 64+len(msg)]`, so a long `info` makes the two overlap and the hash tramples its own input. The guard rejects the moment they would overlap — rejecting beats being silently wrong.

What is checked — RFC 4231 and RFC 5869 standard vectors, VM/native agreement. **Not built** — constant-time guarantees, a MAC comparison op, multi-block HKDF-Expand (L > 32), hashes other than SHA-256, streaming MACs. See also — `hash` (SHA-256 itself) · `aead` (where the split keys are used) · `x25519` (where the secret is made).

chacha — ChaCha20 stream cipher

Source	`lib/chacha.low`
Layer	L0 — pure computation
Capabilities	none

Makes a **keystream** from key and nonce and XORs it with the input (RFC 8439 §2.4). It is symmetric, so the same call encrypts and decrypts.

> **This alone is not safe**
>
> ChaCha20 **only hides** — it does not stop forgery. Flip a bit of ciphertext and the same bit of plaintext flips, unnoticed. It must be used together with authentication, and that place is `aead`. Without a special reason, use `aead`. **Nonce reuse is fatal** — running the same (key, nonce) twice reveals the XOR of two plaintexts. It does not promise constant time and has not been audited.

It is pure computation using only 32-bit addition, XOR and rotation, so it is expressible in the language — VM/native cross-checks and contracts apply as they are. **It does 32-bit arithmetic on `u64`.** This language's `add` stops on overflow, so places needing wraparound state the mask **explicitly** (`bit_and … 4294967295`). The discipline "if you want wrapping, say it by name" pays off directly in cryptography — silent wrapping is a silent defect.

op	What it does	Requires
`prep`	Sets up 16 words of state from key, nonce and counter	key $\geq 32 \cdot$ nonce ≥ 12
`rounds`	Runs 20 rounds (10 double rounds)	work is 16 words
`keystream`	Produces a 64-byte block from the state	ks ≥ 64
`stream`	Runs the whole input and writes to `out`	out $\geq$ len(inp)

Table 50.1 — Ops of `chacha`

The face usually used is `stream`. It answers the number of bytes processed, and **0 means failure**.

What is checked — the RFC 8439 §2.3.2 block function vector and §2.4.2 encryption vector. The block function is measured separately because matching only the stream can coincide even with a wrong state layout, and then the next person to build on it goes wrong. **Not built** — authentication (`aead`), XChaCha20 (extended nonce), counter exhaustion detection, constant-time guarantees.

`poly` — Poly1305 one-time authenticator

Source	`lib/poly.low`
Layer	L0 — pure computation
Capabilities	none

Feeds a message in 16-byte blocks and produces a **16-byte tag**. It testifies that "someone who knows this key sent this byte string" (RFC 8439 §2.5).

> **The key must be fresh for every message**
>
> Poly1305 is a **one-time** authenticator. Authenticating two messages with the same key lets the key be recovered, after which forgery is possible. That is why `aead` makes a one-time key from the nonce for every message (`key_gen`). If you use this module directly, you carry that discipline yourself. It does not promise constant time (there are bounds checks and stops) and has not been audited. Comparing tags is the caller's job — compare without early return (as `aead.unseal` does).

It does 130-bit arithmetic on `u64` split into **five 26-bit limbs**. The largest product is around 2^{52}, safe inside `u64` — **the limb layout is the safety argument**, which is why that number is written at the top of the source.

op	What it does
`setup`	Sets up state (five h · five r) from the key
`block`	Feeds one 16-byte block. `addhi` is 1 for a full block
`emit`	Produces the 16-byte tag from state and key

Table 50.1 — Ops of `poly`

`addhi` is an argument because a short last block must place the top 1 bit differently. `aead` **fills every block** with `pad16`, so it always passes 1 — the saying that padding exists to reduce cases to one becomes concrete here.

`block` stays as this module's **definition of the meaning**. The loop over blocks, though, is what `aead` now calls as the word `poly1305` — the state has the same layout (five h, five r), so `setup` and `emit` remain in the language and only the middle went down (measured 331 → 1840 MB/s). A regression test holds the word and the loop over `block` against the same state.

What is checked — the RFC 8439 §2.5.2 tag vector and VM/native agreement. **Not built** — constant-time guarantees, tag comparison, detection of key reuse, serialising streaming state.

aead — ChaCha20-Poly1305 seal and open

Source	`lib/aead.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Hides a message (`chacha`) and **testifies to** the message and associated data **together** (`poly`). It is exactly what TLS 1.3's default suite (`TLS_CHACHA20_POLY1305_SHA256`) uses (RFC 8439 §2.8). **If you use cryptography, sealing starts here.**

> **What this implementation cannot promise**
>
> **It is not constant time.** This code does not branch on secrets, but the language has bounds checks and stops, so timing cannot be promised. Do not use it as is where a remote attacker can measure time. **It has not been audited** — it is checked against RFC 8439 §2.8.2 vectors, and the guarantee goes as far as "the value the standard specifies comes out". **Reusing a nonce is the end** — two uses of the same nonce with the same key overlap the streams, reveal plaintext and allow tag forgery. This module does not prevent it — counting is the caller's job.

It receives no capabilities — it works on the key and nonce it **is given**. Sealing and opening allocate nothing themselves, hence many arguments: result places (`ct`, `msg`, `tag`) and workspaces (`otk`, `st`, `work`, `ks`, `pst`, `pad`) are all passed in.

op	What it does	Answers
`seal`	Seals `msg` into `ct` and produces a 16-byte `tag`	tag length 16. **0 means failure**
`unseal`	Checks `tag` first and gives `msg` **only if it matches**	plaintext length. **0 means failure (or forgery)**
`key_gen`	Makes a one-time Poly1305 key per nonce	32

Table 50.1 — Ops of `aead`

- `seal` starts the stream at counter **1** — counter 0 was already used for the Poly1305 key. If that one slot overlapped, the tag key would equal the plaintext stream.
- The array fed to the MAC is `AAD ‖ pad16(AAD) ‖ CT ‖ pad16(CT) ‖ le64(|AAD|) ‖ le64(|CT|)`. Thanks to `pad16` every block is a full 16 bytes, and internal calls all share one shape.
- **Failure is 0, the same value as forgery.** `unseal`'s 0 means "the tag does not match" or "a buffer is too small". They are not told apart because the caller must treat both alike — either way, the plaintext must not be used. Code that receives 0 and carries on is already wrong.
- Tag comparison **accumulates XOR** — no early return. It is the best this language can do, and it is also the limit the warning above speaks of.

What is checked — RFC 8439 §2.8.2 vectors and **tamper rejection** (flip one bit of ciphertext or AAD and `unseal` must answer 0), VM/native agreement. **Not built** — constant-time guarantees, streaming sealing, nonce management and counter exhaustion detection, XChaCha20, key derivation (HKDF of `hmac`). See also — `x25519` (key agreement).

x25519 — key agreement on a curve

Source	`lib/x25519.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

From my secret scalar and the peer's public point, makes **32 bytes only the two of us know** (RFC 7748). Standard practice is not to use those bytes as a key directly but to split keys out with HKDF from `hmac`.

> **What this implementation cannot promise**
>
> **It does not promise constant time.** The ladder is written with conditional swaps (`fsel`) so as not to branch on secret bits, but the language has bounds checks and stops, so timing cannot be promised. **It has not been audited. It does not filter small-order points** — the public key a peer sends may be a point that yields zero, so checking whether the agreed result is all zeros is the caller's job. Making private keys (entropy) is not here — that belongs to `random` and `cap random`.

```
rem w has at least 175 u64 elements --- one workspace
let n u64 be x25519.agree shared mysecret theirpub zbuf w .
guard eq n 32 . else return 1 .
```

Why one workspace. `scalarmult` takes nine field elements and the multiplication area separately — easier to read inside. But laying out fourteen at the call site quickly eats the parameter limit of 16. The first program using this library was actually rejected with `E-IR-ARITY`. So `agree` slices one `w` (at least 175 elements), and the layout (nine field elements = `w[0..144]`, multiplication area = `w[144..175]`) lives in one place in the source.

How the representation was chosen. The prime is `p = 2^255 - 19`. The common reference implementation (ref10) holds limbs of 26 and 25 bits, **lets limbs go negative** and carries with an arithmetic right shift. This language's `shr` is a **logical shift**, so that layout cannot be used. So a **representation with no negatives at all** (16 limbs × 16 bits, the TweetNaCl layout) was chosen. Subtraction is `a + 4p - b` instead of `a - b` — since `4p ≡ 0 (mod p)` the value is the same and every intermediate is non-negative. The tool's limit chose the representation, and the one chosen was easier to reason about.

op	What it does	Requires · answers
`agree`	scalar × point → 32-byte shared secret (one workspace)	`w ≥ 175` · others 32 bytes each. Answers 32, **0 means failure**
`scalarmult`	The same computation, workspaces separately	fourteen arguments
`fzero` · `fone` · `fcopy` · `fcar`	Field element init · copy · carry	for code that knows the layout
`fadd` · `fsub` · `fmul` · `fsq` · `fmul121665`	Field add · subtract · multiply · square · constant multiply	ditto

`finv` · `fsel` · `funpack` · `fpack`	Inverse · conditional swap · bytes ↔ element	ditto

Table 50.1 — Ops of `x25519`

Field arithmetic is `export` **for testing** (identities are measured). Unless you are building a new protocol, `agree` is all you need. `ed25519` reuses the same field.

What is checked — RFC 7748 §5.2 scalar multiplication vectors, the §6.1 DH round trip (do two parties reach the same secret), field identities (`a·a⁻¹ = 1` and so on), VM/native agreement. **Not built** — constant-time guarantees, small-order point filtering, key generation and clamping convenience ops, X448.

aes — the AES-128 block cipher

Source	`lib/aes.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Scrambles 16 bytes with a key and yields 16 bytes (FIPS 197). That is all — the length does not change, there is no authentication, and the same input always gives the same output. It exists because TLS 1.3 makes `TLS_AES_128_GCM_SHA256` a MUST (RFC 8446 §9.1).

> **Not a module to use alone**
>
> The same plaintext block becomes the same ciphertext block, so patterns show through, and alterations to the ciphertext go unnoticed. What you actually need is `encrypt` and `decrypt` of `gcm` this module is the part beneath it. It does not promise constant time and has not been audited. It is a pure Lowent implementation without AES-NI and is slow — if you need throughput, consider `aead` (ChaCha20-Poly1305) first. Keys are 128 bits only.

op	What it does	Requires
`gmul`	GF(2^8) product (irreducible polynomial `0x11b`)	—
`ginv`	GF(2^8) inverse — `x^254` (Fermat)	the inverse of `0` is `0`
`sbox` · `inv_sbox`	One S-box byte — **computed from the definition**, not a table · inverse S-box	—
`expand_key`	16-byte key → 176 bytes of round keys	`rk ≥ 176` · `key ≥ 16`
`encrypt_block`	Encrypts 16 bytes of `st` in place	`rk ≥ 176` · `st ≥ 16` · `tmp ≥ 16`

Table 50.1 — Ops of `aes`

There is no block decryption. GCM is counter mode and uses encryption only — decryption is done with encryption too. `inv_sbox` existing means a place was left for it someday, not that it was built.

The S-box is not written as a table. Instead of transcribing 256 constants by hand, the definition `S(x) = affine(x^-1 in GF(2^8))` is used directly. Without a table to transcribe there is nothing to transcribe wrongly. A test compares the 256 computed values with the canonical table.

Performance — as measured. In the development repository, one `sbox` took 180 ns (7.5 ns if read from a table), one `encrypt_block` about 31 µs, 93 % of it the 160 `sbox` calls. Whole AES-128-GCM ran at 0.46 MB/s; ChaCha20-Poly1305 at 41.8 MB/s. A factor of 15 has already been repaid — `ginv` was multiplying `x^254` 254 times; switching to square-and-multiply took it from 254 to 15 multiplications, and not one digit of the test vectors moved. A table would bring GCM to about 2.5 MB/s, still 17 times slower than ChaCha20, and a table brings in a **secret-indexed cache side channel** — AES's classic attack surface. We do not buy a side channel for a slow fallback.

What is checked — FIPS 197 Appendix B and C vectors, the record ciphertext of RFC 8448 §3 (used end to end), VM/native agreement.

`gcm` — AES-128-GCM authenticated encryption

Source	`lib/gcm.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Gives "nobody can read it" and "nobody can alter it" **at once** (NIST SP 800-38D). `aes` gives only the first.

> **What it promises and what it does not**
>
> It does not promise constant time and has not been audited. **Never repeat a nonce** — using the same nonce twice with the same key makes GCM **lose both the plaintext and the authentication key**. It is the most expensive mistake in this mode. **It is slow** — measured at 0.46 MB per second in the development repository, far slower than ChaCha20-Poly1305 on the same machine (41.8 MB). So the preferred suite is ChaCha20. This module exists **correctly** because the standard makes it a MUST, not to be fast. A tag check failure in `decrypt` is a value — always look at the return value.

What is AAD. Bytes that are not hidden but **must not be altered**. In TLS 1.3 it is a record's 5-byte header — an observer sees the length anyway, but if the peer changes it the tag must not match. Unlike `seal` and `unseal` of `aead`, this module uses the names `encrypt` and `decrypt`. There is a defect where, if two modules share a name, the processor measures the argument count against the other module's signature even when called qualified, so the names were separated to avoid it.

op	What it does	Requires
`gmul128`	GF(2^{128}) multiplication of GHASH — `z ← z·h`	`z ≥ 16 · h ≥ 16 · v ≥ 16`
`encrypt`	`msg` → ciphertext and a 16-byte tag	`nonce = 12 · key = 16`
`decrypt`	**Checks the tag first** and gives the plaintext	on failure answers 0 and gives no plaintext

Table 50.1 — Ops of `gcm`

The caller holds the backing (`scratch`). If short it answers 0 and writes nothing — guards keep that.

GHASH bit order — where people slip. The reduction polynomial of GF(2^{128}) is `x^128 + x^7 + x^2 + x + 1`, but GCM uses **reversed bit order** — the top bit of a block's first byte is x^0. So multiplication runs with right shifts, and overflowing bits come back into the first byte as `0xE1`. Read that convention backwards and the ciphertext is right but the whole tag differs, looking only like "decryption fails".

On nonces, again. GCM is counter mode, so the same (key, nonce) yields the same keystream. The XOR of two plaintexts is revealed, and worse, an equation arises from which the GHASH authentication key can be solved, letting the peer forge any message. TLS 1.3 avoids this by **deriving the nonce from the sequence number** (`record_nonce` of `tls13`). If you use it directly, follow that method — random 96-bit nonces collide sooner than you think.

What is checked — NIST GCM test vectors, the record ciphertext of RFC 8448 §3 (ciphertext and tag byte for byte), VM/native agreement.

`crypto_hw` — arithmetic the machine helps with

Source	`lib/crypto_hw.low`
Layer	L0 — pure computation (the caller's scratch)
Capabilities	none

The parts of cryptographic arithmetic **the CPU has an instruction for**. Today that is one thing: multiplication in GF(2^{128}) — what GCM's GHASH stands on, and what CRC is too.

> **There is no `unsafe` in this module**
>
> The **processor** emits the instruction; this source uses ordinary words (`clmul_lo`, `clmul_hi`). So nothing travels: callers stay `effects none`. Writing assembly directly would carry `unsafe` all the way up to TLS (see «The absorbing boundary» in the chapter 30 chapter).

op	what it does	requires
`clmul128`	one carry-less product — 128 bits as two words	`out ≥ 2`
`gf128_mul`	GF(2^{128}) product — GCM's reflected order, Karatsuba (three multiplies)	`out`, `x`, `h` each `≥ 2`

Table 50.1 — ops of `crypto_hw`

What carry-less multiplication is. Multiplication with no carries — addition is xor. Every output bit is a combination of input bits, which is why cryptography and checksums are built on it. The answer is 128 bits and this language has no 128-bit type, so it comes back as **two words** (low, high).

The build decides the speed. Compiled with `lowentc --hw auto`, it uses the instruction where the machine has one and the plain computation where it does not. With `--hw none` (the default) it is always the computation. **The answer is the same either way** — only speed and timing behaviour differ.

Measured. The leaf `ghash` went 82 → 6562 MB/s, the leaf `aes_ctr` 80 → 5475, and AES-128-GCM as a whole 41 → 2848 MB/s (4 MiB, gcc, best of each column across three alignment settings). Those numbers need the AES instructions turned on too (`--hw aes`): with only one of the two, the bottleneck just moves.

Carrying the instruction was not the end of it. With the same instruction, the first version drew only half of what the machine could give. The other half was in **how it was used** — eight blocks at a time, one reduction per eight rather than per block, counters built in a register instead of through memory, and the reduction itself cut down to two multiplies. The leaf `aes_ctr` now runs at **the same speed** as OpenSSL on this box.

What is checked — the golden suite compares this module's `gf128_mul` against the processor's own leaf `ghash`: the same product computed in two different places, each the other's witness. It also compares the instruction path, the computed path and the VM on the same inputs.

> **Constant time is not promised**
>
> The instruction path reads no tables, so its timing behaviour is better. Even so, this repository's cryptography promises no constant time and has not been audited — other places remain. Saying so is how that promise stays honest.

`bigint` — big-number modular arithmetic

Source	`lib/bigint.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Modular multiplication and exponentiation for numbers that do not fit in one `u64` — such as RSA's 2048-bit numbers.

What it promises and what it does not

It does not promise constant time and has not been audited. **Only what is needed was built** — what RSA verification (`s^e mod n`) and curve arithmetic require. There is no big-number division, no negatives, no GCD, no inverse. The exponent `e` is **one `u64`** — narrowed to hold RSA's public exponent (usually 65537). What was not built cannot be wrong.

Representation — the limb width is set by accumulation. Numbers are **16-bit limbs** held in `u64`, laid out little-endian. A product is 16×16 = 32 bits, and adding 128 of them stays at 2^{39}, safe inside `u64`. With 32-bit limbs the product reaches 2^{64} and accumulation overflows. "The register is 64 bits, so limbs are 64 bits" looks natural but is wrong — the limb width is set not by the product but by accumulation.

op	What it does	Requires · notes
`zero`	Zeroes the first `k` limbs of `a`	`a ≥ k`
`from_bytes` · `to_bytes`	**Big-endian** bytes ↔ limbs	RSA's encoding
`from_bytes_le` · `to_bytes_le`	**Little-endian** bytes ↔ limbs	curve convention
`ge_mod`	Is `a ≥ n` (0 · 1)	compares `k` limbs
`dbl_mod`	`a ← 2a mod n`	used to make R^2
`mont_mul`	`out ← a·b·R⁻¹ mod n` (CIOS)	`t` is workspace
`n0inv16`	$-n^{-1} \bmod 2^{16}$ — the Montgomery constant	takes only the lowest limb of `n`
`mod_exp`	`out ← bse^e mod n`	`e` is `u64` · `r2`, `acc`, `tmp`, `t` are workspace

Table 50.1 — Ops of `bigint`

Why Montgomery — to avoid building division. In the usual modular product `a·b mod n`, the `mod` is big-number division. Montgomery multiplication yields $a \cdot b \cdot R^{-1} \bmod n$ instead, and with $R = 2^{16k}$ the division becomes a **shift**. The price is moving values into and out of the Montgomery domain and needing $R^2 \bmod n$, which is also obtained without division — doubling 1 $2 \cdot 16k$ times (`dbl_mod`) and subtracting `n` whenever it overflows. The whole design of this module is that one line: **the product was changed so that division need not be built.**

What is checked — byte comparison with Python integer arithmetic, RSA-PSS verification passing end to end (`rsa`), VM/native agreement. **Not built** — constant time, big-number division, negatives, GCD, modular inverse (the curve side's inverse is Fermat exponentiation in `p256`), very large `e`.

`rsa` — RSASSA-PSS verification

Source	`lib/rsa.low`
Layer	L0 — pure computation
Capabilities	none

Answers whether an RSA-PSS signature from a peer matches the public key and message (RFC 8017 §8.1.2). TLS 1.3's `rsa_pss_rsae_sha256` is this.

> **What it promises and what it does not**
>
> It does not promise constant time and has not been audited. It does **PSS only** — PKCS#1 v1.5 is not built, because RFC 8446 §4.4.3 forbids it in CertificateVerify. There is no SHA-1 family, key generation or decryption (RSA encryption). Signing is not built either — a TLS 1.3 server may need it someday, but what came first is `p256`, with smaller keys and signatures.

The old PKCS#1 v1.5 had deterministic padding and shallow structure, and attracted several attacks. PSS mixes in a salt and covers it with a mask. PSS verification has three pieces.

```
① s^e mod n  →  EM (encoded message, emLen = key length)
② make a mask with MGF1(SHA-256) and strip DB
③ does H' = SHA-256(0x00×8 ‖ mHash ‖ salt) equal H inside EM
```

① is `mod_exp` of `bigint`, and ② and ③ are this module's job. The eight `0x00×8` bytes are not decoration — they are a prefix the standard added to separate signed content from certificate signatures. Leaving them out lets signatures from another context be reused.

op	What it does	Requires
`mgf1`	The MGF1-SHA256 mask generation function	`out ≥ n · w` is backing
`pss_verify`	Given EM, is the PSS encoding right	only ② and ③
`verify_pss`	signature · modulus · exponent · hash → 0 or 1	all of ①②③

Table 50.1 — Ops of `rsa`

Usually only `verify_pss` is used. `pss_verify` is exposed separately to test ① apart from ② and ③.

What is checked — standard PSS test vectors (positive), two negatives (one bit of signature · one bit of hash), MGF1 against a reference implementation, and VM/native agreement. There are two negatives because changing the signature and changing the message are different failures — checking only one cannot filter out an implementation that lets the other pass.

p256 — the NIST P-256 curve and ECDSA verification

Source	`lib/p256.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Point arithmetic on the elliptic curve `y² = x³ − 3x + b`, and **ECDSA signature verification** built on it. TLS 1.3's `ecdsa_secp256r1_sha256` uses this curve.

What it promises and what it does not

It has not been audited. The verification `smul` **branches** on scalar bits — in verification that scalar is public, so that is fine. For secret scalars (private key, nonce) use `smul_ct` — each step does both doubling and addition and chooses by mask, not by branch. That promise extends to **no secret-dependent branches or memory access**; cache hierarchy, power and electromagnetic leakage are not promised. There is no point decompression and no P-384 or P-521.

Why this curve. What public CAs actually issue and browsers actually accept is P-256 and RSA. `ed25519` is mathematically cleaner but does not work on the public web — the ecosystem decides, not the maths. Signature **generation** is done by `ecdsa` on top of this module.

op	What it does
`limbs`	Limb count (16 = 256 bits)
`msub · madd`	Modular subtraction · addition
`minv`	Modular inverse (Fermat exponentiation)
`pmul`	Montgomery product (reusing `bigint`'s)
`make_r2`	`R² mod p` — the constant for entering the Montgomery domain
`pdbl · padd`	Jacobian point doubling · point addition
`is_zero`	Is it the point at infinity
`smul`	Scalar multiplication `[k]P` — for public scalars
`smul_ct`	Scalar multiplication — for secret scalars, choosing by mask without branches
`ecdsa_verify`	Does signature `(r, s)` match hash `e` and the public key

Table 50.1 — Ops of `p256`

Curve constants (p · n · Gx · Gy) are passed in by the caller.

Why Jacobian coordinates — to postpone the inverse. Point addition in affine coordinates needs a modular inverse every time, and an inverse costs hundreds of times a product. Jacobian coordinates carry the denominator inside the coordinates and postpone the inverse to a single one at the end. For the same reason `ed25519` uses extended coordinates, but its addition formula is **complete** with no exceptional branch, and here there **is** one (a different formula when the points are equal). That branch lives inside `padd`.

What is checked — NIST CAVP P-256/SHA-256 signature verification vectors, 2 positive and **8 negative**, field and scalar multiplication compared with a Python reference curve, VM/native agreement. A signature verification test without negatives cannot catch "always true" — a verification test with only positives verifies nothing.

p384 — the NIST P-384 curve and ECDSA verification

Source	`lib/p384.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

ECDSA signature verification on the curve `y² = x³ − 3x + b` over the prime `p = 2^384 − 2^128 − 2^96 + 2^32 − 1`. TLS 1.3's `ecdsa_secp384r1_sha384` uses this curve. Many intermediate certificate authorities on the public web, and their roots, sign with P-384, so checking a certificate chain to its end needs this curve.

What it promises and what it does not

It has not been audited. It exports **verification** only — no signing, no point decompression, no constant-time scalar multiplication. The scalars in verification are public, so branching on them is fine; handling secret scalars is not this module's job.

Why a new module instead of changing p256. `p256` hard-codes its 16 limbs into loop bounds and workspace offsets, and several modules already use it, fully checked. Making it generic over the limb count would touch checked code. So each curve gets its own module — this file is `p256` carried from 16 limbs to 24, and **not one formula differs**. Only the sizes do.

place	p256	p384
one number (16-bit limbs)	16	24
one point (Jacobian X · Y · Z)	48	72
Montgomery scratch	18	26
exponent bits	256	384

Table 50.1 — `p256` and `p384` differ only in sizes

op	what it does
`ecdsa_ok`	does the signature `(r, s)` match hash `e` and public key `pub` — `ok 1` yes · `ok 0` no · `error short_workspace` the workspace is too small

Table 50.2 — ops of `p384`

`ecdsa_ok` takes the curve constants (`p` · `n` · `Gx` · `Gy`) from the caller, the public key in affine form `x ‖ y` (48 limbs), and a workspace `w` of at least 1364 limbs (1.5 × `p256`'s 924). **A short workspace is a failure, not an answer** — once "not enough room" and "bad signature" were both 0, and a program that passed the wrong workspace looked like it had a bad signature.

The computation: first check that `r` and `s` lie in `1 … n−1`, then

```
w  = s⁻¹ mod n
u1 = e·w mod n ,  u2 = r·w mod n
R  = u1·G + u2·Q          (Q = the public key point)
true if R.x ≡ r (mod n)
```

Modular multiplication reuses `bigint`'s Montgomery product — `p` and `n` are both odd, so the condition holds and no new arithmetic is built. Coordinates are Jacobian for the same reason as `p256` (the inverse is postponed to one at the very end).

What is checked — two oracles we did not build. ① A signature made by `openssl` (secp384r1 · SHA-384) and accepted by `openssl dgst -verify` — one positive and **three negatives** (one bit of `s` · one bit of the hash · `s = 0`). ② Real certificate chains signed by real CAs (the intermediate of several public sites is P-384). The NIST CAVP vectors have **not been run yet** — the file is not in the repository. The consumer of this module is `verify`.

ecdsa — ECDSA P-256 signing

Source	`lib/ecdsa.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Takes a private key and a message hash and produces a signature `(r, s)` (RFC 6979). Verification is `ecdsa_verify` of `p256`. A TLS server uses it to sign `CertificateVerify` with its own private key — without signing there is no server.

> **What it promises and what it does not**
>
> **Constant time is promised here only, with its extent written down.** What is removed is branching and memory access that depend on secret scalars (private key d, nonce k). The final conditional subtraction of Montgomery multiplication, the cache hierarchy, power, electromagnetic leakage and whatever a compiler might introduce are not promised. It has not been audited. P-256 only, no key generation. If `r = 0` or `s = 0` it does not retry but answers failure (probability around 2^{-128}) — building that branch creates untestable code, and untested cryptographic code is worse than none.

Why the nonce is derived, not random. In ECDSA, a nonce leaked once or repeated once reveals the whole private key.

$s = k^{-1} (h + r \cdot d) \quad \Rightarrow \quad d = (s \cdot k - h) / r$

Knowing `k` finishes it in that one line, and using the same `k` twice lets `k` be solved from two signatures. And the ways a random source fails are silent — empty entropy, identical state after a fork, a restored virtual machine snapshot. So **no random source is used at all.** `k = HMAC-DRBG(private key, message hash)`. The same (key, message) gives the same signature, and that is a property, not a defect.

op	What it does	Requires
nonce6979	RFC 6979 §3.2 — (d, h) → k	`w ≥ 480` bytes · `wu ≥ 16` limbs
sign	(d, h) → `r ‖ s`, 64 bytes	`wb ≥ 512` bytes · `wu ≥ 908` limbs

Table 50.1 — Ops of `ecdsa`

Curve constants (p · n · Gx · Gy) are given by the caller — this module holds no tables. Signing uses only `smul_ct` of `p256`, and a test confirms it. k^{-1} is Fermat exponentiation whose exponent (n − 2) is public, so the order of operations is fixed.

Generosity is not free. The workspace was first sized at 1052 limbs, which exceeded `--run`'s array argument limit (u64 1024) and could not be tested on the VM. Gaps were pulled in twice to reach 908 — a size that cannot be tested is not a size but a defect. **Performance** — `sign` has many parameters, locals and instructions and does not enter the typed fast path. One signature per connection makes that bearable for now.

What is checked — the official RFC 6979 §A.2.5 vectors (nonce k and r · s for messages `"sample"` and `"test"`, byte for byte), whether our verifier accepts the signatures produced, and whether flipping one bit is rejected. Vector comparison asks "is it per the standard"; the round trip asks "do our two sides agree".

ed25519 — Ed25519 signature verification

Source	`lib/ed25519.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Verifies signatures on the twisted Edwards curve over **the same prime** as Curve25519, `p = 2^255 - 19` (RFC 8032 §5.1.7). Because the prime is the same, the field arithmetic of `x25519` is reused as is — same prime, same field. That one does key agreement; this one verifies signatures.

What it promises and what it does not

It does not promise constant time and has not been audited. **There is no signing.** No `ed25519ph`, `ed25519ctx` or Ed448. **This algorithm cannot be used for public web HTTPS server certificates** — public CAs practically do not issue Ed25519 server certificates. That is `p256`'s place. This module verifies **a peer's** signatures.

Edwards addition is complete. It uses extended coordinates `(X:Y:Z:T)`, `xy = T/Z`. The Edwards addition formula has no exceptions — unlike Jacobian there is no "different formula when the points are equal". The branch inside `p256`'s `padd` does not exist here at all, and without a branch nothing can go wrong in it or leak through its timing.

Ambiguous input is rejected. RFC 8032 lets implementations differ in handling non-canonical encodings and small-order points. Here they are rejected — `S < L` is enforced (otherwise one message has several valid signatures), and failed decompression is rejected. When a standard says "either is fine", writing down the choice is the document's job.

op	What it does
`verify`	Does signature `sig` match public key `pub` and message `msg` — usually the only one used
`make_d` · `make_l` · `make_base`	Curve constant `d` · order `L` · base point `B`
`eadd` · `esmul`	Point addition · scalar multiplication
`compress` · `decompress`	Point ↔ 32-byte encoding
`fpow` · `feq` · `fbit0`	Field exponentiation · equality · lowest bit
`pow2_minus` · `reduce_l`	`2^n - k` · reduction `mod L`

Table 50.1 — Ops of `ed25519`

The verification equation is `[S]B = R + [k]A` with `k = SHA-512(R ‖ A ‖ M) mod L`.

Constants that must be transcribed are checked by a property. `d`, `√-1` and `B` are computed — nothing to write down. The order `L` cannot be derived and had to be written. So a test checks that `[L]B` is the identity. Write it wrong and that check breaks — a way to measure a transcription instead of trusting it.

What is checked — RFC 8032 §7.1, 3 positive and **6 negative**, `[L]B = identity`, VM/native agreement.

`der` — minimal DER parser

Source	`lib/der.low`
Layer	L0 — pure computation
Capabilities	none

Tells **where the public key starts and how many bytes it is** in X.509 certificate bytes, and finds the EC scalar in a PKCS#8 private key file. Verifying signatures with that key is the job of `p256`, `rsa` and `ed25519`.

> **This is not PKI**
>
> Chain validation, CA trust stores, name matching, validity periods, revocation, extensions — **it does none of them**. So this alone cannot answer "can this certificate be trusted". The caller must **pin** the public key or judge by its own rules. A half-built PKI is worse than none — it makes you believe you can trust it.

Lengths were written by the other side. Every DER length field is chosen by the attacker. So this parser keeps three rules — any length exceeding the buffer fails immediately (answers 0); long-form lengths are accepted up to 4 bytes only; there is no recursion (depth is a loop the caller controls). **Every failure is `0`** — a value, not an exception.

op	What it does
`tag_at`	Tag byte at `off`
`value_off` · `value_len`	Where that TLV's **value** starts · its length
`next_off`	Position of the next sibling TLV
`is_spki`	Is this position **shaped** like SubjectPublicKeyInfo
`find_spki`	Finds the SPKI inside a certificate
`alg_oid_off` · `alg_oid_len`	Position · length of the algorithm OID
`key_off` · `key_len`	Position · length of the public key bit string
`rsa_n_off` · `rsa_n_len` · `rsa_e`	Modulus and exponent inside an RSA key
`p8_alg_off` · `p8_inner_off` · `p8_inner_len`	PKCS#8 algorithm · inner key position · length
`ec_priv_off` · `ec_priv_len`	Position · length of the EC private key (scalar)

Table 50.1 — Ops of `der`

Found by shape, not counted by position. The SPKI is the sixth or seventh field of a certificate — depending on whether `version` is present. Counting positions goes silently wrong on old certificates. So `is_spki` looks at the shape `SEQUENCE { SEQUENCE { OID … }, BIT STRING }`. Counting breaks when what comes before changes; recognising does not.

What is checked — parse results are not eyeballed. The key is extracted from the real certificate of RFC 8448 §3 and used to **verify** that handshake's CertificateVerify signature. If parsing slips by one byte the signature does not match. Two negatives (one bit each of signature and transcript) are checked too. A check where a person looks at whether the extracted value seems plausible is not a check.

`pem` — unwrapping PEM envelopes

Source	`lib/pem.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Extracts **the bytes in the middle (DER)** from the folded base64 between `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` (RFC 7468). Key files have the same shape with a different label. `codec` knows base64 but not line folding — this module strips the header, unfolds and hands it on. Certificates come from outside (a tool like certbot leaves them as files). ACME is not built.

What it promises and what it does not

It cannot open encrypted PEM (`Proc-Type: 4,ENCRYPTED`). **It yields only the first one** — if a file holds several (a certificate chain), the caller calls again for the later ones. No URL-safe base64. **It is a parser, not a trust decision** — it verifies nothing.

op	What it does
`find_from`	Finds `needle` in `hay` (`len hay` if absent)
`body_off`	Position of **the line after** `-----BEGIN <label>-----`. 0 = absent
`end_off`	Position of `-----END <label>-----`. 0 = absent
`unwrap`	One PEM block → DER bytes. `option u64` (bytes written), `none` on failure

Table 50.1 — Ops of `pem`

In `unwrap src label scratch out`, `scratch` carries two loads — assembling the header (front) and the unfolded base64 (back). `len scratch ≥ len src + label length + 16` is enough.

Why a label is required. `unwrap` takes by name what it opens, and rejects when the BEGIN and END labels differ. A file holding several blocks is normal. Without matching pairs, opening `CERTIFICATE` would swallow up to the next BEGIN and decode garbage — and base64 silently accepts garbage too.

Line ends accept both LF and CRLF — the opposite of `http`. There, the boundary is security, so a bare LF is rejected (request smuggling). Here it is a file format, and real files use both. Strictness is not a virtue but a tool — its strength depends on what it guards against.

Why `unwrap`, not `decode`. `utf8.decode` already exists, and when names collide there is a known defect where the processor measures types with another module's signature even when called qualified. A test using both modules together caught that combination breaking. A library green on its own is not enough — it must compose to be usable.

The path PEM → DER → PKCS#8 → 32-byte scalar goes with `der` — get DER with `pem.unwrap src "PRIVATE KEY" sc buf`, then find the scalar's position with `der.p8_inner_off` and `der.ec_priv_off`.

What is checked — comparison with openssl output (certificate DER of 375 bytes identical, the scalar inside PKCS#8 at position 36 with length 32), rejection of label mismatch and missing envelope, VM/native agreement. The test vector's scalar is the synthetic value `01 02 … 20` — the structure is exactly what openssl produced, so it measures the parser just as well, and it is plainly not a secret.

`x509` — reading X.509 certificates

Source	`lib/x509.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Reads the skeleton of an X.509 certificate (DER): where the signed part starts and ends, who the issuer and subject are, and where the public key, validity and extensions sit — all as **offsets and lengths**. It is the step after `der`, which stopped at "extract the public key".

> **It only reads**
>
> This module **does not check** a certificate. Whether the signature holds, whether a chain links, whether a name matches — that is `verify`, built on top of this one. Not built: revocation (CRL · OCSP) · name constraints · policies · the old v1/v2 shapes · comparing names that are not UTF-8.

The shape of a certificate and where this module's ops point:

```
Certificate
├─ tbsCertificate ──────────────── tbs_off … tbs_end   ← the bytes the signature covers
│   ├─ [0] version (may be absent)
│   ├─ serialNumber                tbs_field 0
│   ├─ signature                   tbs_field 1
│   ├─ issuer (issuer name)        issuer_off · elem_len
│   ├─ validity                    not_before · not_after
│   ├─ subject (subject name)      subject_off · elem_len
│   ├─ subjectPublicKeyInfo        spki_off
│   └─ [3] extensions              ext_value_off · is_ca · san_next
├─ signatureAlgorithm ──────────── sigalg_oid_off · sigalg_oid_len
└─ signatureValue ──────────────── sig_off · sig_len
```

op	what it does
`tbs_off` · `tbs_end`	start and end of the signed bytes. A signature is checked over **these original bytes**
`sigalg_oid_off` · `sigalg_oid_len`	offset and length of the outer signature algorithm OID
`sig_off` · `sig_len`	the signature value (past the BIT STRING's "unused bits" byte)
`tbs_field`	offset of the `n`-th field in tbsCertificate (0 serial · 1 signature · 2 issuer · 3 validity · 4 subject · 5 public key)
`issuer_off` · `subject_off` · `elem_len`	offset and length of the **whole** issuer / subject name element — names are compared as bytes
`spki_off`	offset of the public key (SubjectPublicKeyInfo)
`not_before` · `not_after`	validity as one comparable number (`YYYYMMDDhhmmss`). Only the `Z` time zone is accepted
`ext_value_off`	value offset of the extension whose OID is `2.5.29.<n>` (17 = subject alternative name · 19 = basicConstraints)
`is_ca`	may this certificate issue others. **No extension means false**

`san_next`	the DNS names of the subject alternative name, one at a time (`cur` 0 = the first, 0 back = no more)

Table 50.1 — ops of `x509` — an op that returns an offset returns 0 for “malformed”

No copies. Every op returns only an offset and a length. A few certificates must fit in a 64 KiB arena at once, and a signature must be checked over the original bytes — checking a copy could make a signature look right even when the copy is wrong.

Every length was written by the other side. A certificate is bytes nothing has vouched for yet. Every offset passes through `der.value_off`·`der.value_len`, which keep it inside the buffer. This module adds **order** on top: X.509 fields have no labels, so a field’s position is what it is — ignore the order and you read a field someone slipped in as if it were in its own place.

The default of `is_ca` matters. Without a basicConstraints extension a certificate is not a CA (RFC 5280 §4.2.1.9). Miss that default and a leaf certificate can act as an intermediate — the most common hole in chain checking.

`verify` — certificate signatures and one link of a chain

Source	`lib/verify.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Once `x509` has **read** a certificate, this module answers on top of it: **was this certificate signed by that one**. Besides the signature, it gathers what one **link** of a chain needs — validity, names that connect, and host name matching.

> **What it promises and what it does not**
>
> It has not been audited. It accepts four signature algorithms only — RSA v1.5 + SHA-256 · RSA v1.5 + SHA-384 · ECDSA + SHA-256 · ECDSA + SHA-384. Anything else (the SHA-512 family · RSA-PSS certificates · Ed25519 certificates · P-521) is **refused**. There is no revocation check. Letting an unknown algorithm through would turn this module's answer from "checked" into "could not tell".

A chain is a line of links

A web server usually sends three layers. Each is signed by the one above it, and the top (the root) must be one we **decided in advance to trust**.

```
root (in the trust store)     ← found by trust.find_anchor
   │  signs
   ▼
intermediate CA               ← link_ok intermediate · root
   │  signs
   ▼
leaf (example.com)            ← link_ok leaf · intermediate  +  host_ok leaf · "example.com"
```

Call `link_ok` for each link from the bottom up, and add `host_ok` for the leaf. If any answer is not 1, the chain is broken. **How** the chain is gathered (the order the server sent, how many links) is the caller's choice — `apps/lowget` is a real example.

value	meaning
1	links — signature, CA, names and validity all hold
2	the child's issuer name differs from the parent's subject name
3	the parent is not allowed to issue certificates (not a CA)
4	one of the two is outside its validity period
5	the signature does not hold (or the algorithm is unknown)
6	the workspace for checking the signature was too small — not a result, but **could not measure**

Table 50.1 — what `link_ok` returns — anything but 1 says why it refused

ops

op	what it does
`link_ok`	links one step (table above). Byte workspace ≥ 1024 · limb workspace ≥ 2200
`signed_by`	was the child signed by the parent — `ok 1` / `ok 0`, `error short_workspace` if there is not enough room
`sig_kind`	the algorithm the child names: 1 RSA+SHA-256 · 2 ECDSA+SHA-256 · 3 RSA+SHA-384 · 4 ECDSA+SHA-384 · 0 unknown
`dn_eq`	are two names equal **as bytes**
`dates_ok`	does the validity period contain now (one number `YYYYMMDDhhmmss`)
`host_ok` · `san_matches`	does a host name match one of the subject alternative names
`ecdsa_rs`	takes `r` · `s` out of a DER-wrapped ECDSA signature, right-aligned to the curve size (32 · 48 bytes)
`curve` · `curve384`	fills the P-256 · P-384 constants (`p` · `n` · `Gx` · `Gy`) into a workspace

Table 50.2 — ops of `verify`

Design

Signatures are checked over the original bytes. `x509.tbs_off … tbs_end` is hashed as is. Checking a copy could make a signature look right even when the copy is wrong.

The public key length picks the curve. 65 bytes is P-256 (`p256`), 97 bytes is P-384 (`p384`). RSA goes to `rsa`.

Names are compared as bytes. DER has one byte string per name, so that is right. Decoding and comparing as text lets case, encoding and whitespace rules in, and each rule shifts what "equal" means.

A wildcard covers the first label only. `*.a.b` matches `x.a.b` but not `a.b` or `y.x.a.b` (RFC 6125 §6.4.3). Case folding is ASCII only.

`ecdsa_rs` right-aligns. A DER integer may carry a leading 0 (when the top bit is 1) or be short. Copying it as is verifies a value shifted by one byte, and that mistake only ever shows up as "bad signature".

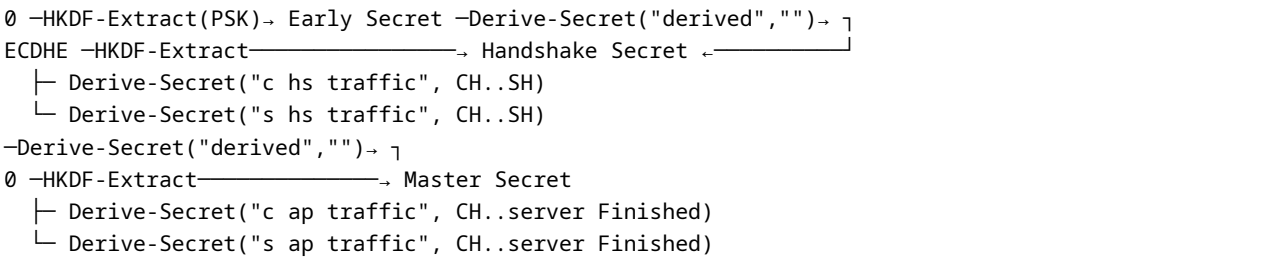
A short workspace is a failure. It is `error short_workspace` — once "not enough room" and "not signed" were both 0, and a program that passed the wrong workspace looked like it had a bad signature.

`tls13` — the computational parts of TLS 1.3

Source	`lib/tls13.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

Four pure computations TLS 1.3 requires (RFC 8446) — the **key schedule** (deriving traffic keys from the shared secret and transcript hash, §7.1), the **record layer** (sealing and opening records, §5), the **transcript hash** (the hash of every message exchanged so far, §4.4.1), and **Finished** (proof that the handshake was not altered along the way, §4.4.4). All are pure, so they can be built without transport and measured against test vectors — parser first, transport later.

> **This is not a TLS implementation**
>
> There is no state machine — it does not drive the handshake (message layer and ordering are `tlssrv`'s). There is no transport. No PSK, 0-RTT, exporter or resumption secrets; only the 1-RTT path was built. No PKI (`der`). It does not promise constant time and has not been audited. There are two suites, and unknown suites are rejected.



Record keys come from each secret — `key = Expand-Label(secret, "key", "", length)`, `iv = Expand-Label(secret, "iv", "", 12)`.

op	What it does
`build_label` · `expand_label`	HkdfLabel structure to bytes · `HKDF-Expand-Label`
`derive_secret`	`Derive-Secret(secret, label, transcript hash)`
`advance`	One rung of the ladder — one arrow in the diagram above
`traffic_key` · `traffic_iv`	Secret → record key · IV
`finished_key` · `verify_data`	Key and value of Finished
`record_header` · `record_nonce`	5-byte header `23 ‖ 0x0303 ‖ length` · IV and sequence number → nonce
`record_seal` · `record_open`	Sealing · opening a record
`inner_type`	Reads the real content type from the **end** of the inner plaintext
`transcript`	Hash of a buffer of concatenated messages
`hs_type` · `hs_size` · `hs_count`	Walking handshake messages
`check_finished`	Recomputes the peer's Finished and compares

Table 50.1 — Ops of `tls13`

A record's outside differs from its inside. The outside is always `23 ‖ 0x0303 ‖ length` — handshake and application data look the same. The real content type is at the end of the inner plaintext, arranged to hide it from observers. The AAD is that 5-byte header itself.

Two suites give negotiation meaning. Suite 1 = `TLS_AES_128_GCM_SHA256` (MUST, `gcm`), suite 2 = `TLS_CHACHA20_POLY1305_SHA256` (SHOULD, `aead`). With only one built, this place was a dead branch. Unknown suites are rejected — it does not silently pick one.

The transcript hash is the backbone of the handshake. Keys and Finished alike stand on the hash of "every message exchanged so far", so if a man in the middle changes even one byte, the two sides' keys diverge. **Where the transcript is cut is half the standard** — RFC 8448 records those points as each Derive-Secret's hash, and tests compare all three (`CH…SH`, `CH…server Finished`, `CH…client Finished`). SHA-256 here works in one shot (no streaming), so the transcript is measured over **a buffer the caller concatenated**. Message walking does not trust length fields — if a length exceeding the buffer is written, it stops there. `check_finished` accumulates XOR over 32 bytes and looks once at the end (no early return).

Why workspaces are bundled into two. The parameter limit is 16, so laying out buffers per suite overflows quickly. So they are bundled into bytes `w` and `u64 u`, with positions written in source comments. It is not pretty — it is the shape the language's limit made, and better not hidden.

What is checked — two layers. §7.1 independently implemented with Python `hashlib` and `hmac` for byte comparison, and 96 values of RFC 8448 §3 extracted mechanically from the canonical text and compared. **And the canonical text caught a defect.** `traffic_key` had the key length fixed at 32, but that length goes inside HkdfLabel, so with a peer using AES-128-GCM (16) the key differs entirely. The second implementation did not catch it — the same person wrote 32 in both. A second implementation is not the canonical source.

`tlssrv` — the TLS 1.3 server handshake

Source	`lib/tlssrv.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

`tls13` holds the **computations** of the standard. This module is the **layer that calls them in order** — reading ClientHello, building server messages, moving state, and sealing and opening application data records. Every failure is `0`, and ops yield **positions** — the caller cuts with `subslice`.

> **This alone is not a TLS server**
>
> **There is no transport** — no sockets, no reassembly; the caller gathers the bytes (and decides record boundaries). No KeyUpdate, record padding, HelloRetryRequest, PSK, 0-RTT, client certificate requests or session resumption. `build_cert` holds only one certificate of a chain. Extensions are only **found** (interpreting SNI and ALPN is the caller's). It has not been audited.

ClientHello is the most hostile input. It is a byte string nothing has authenticated yet, and every length field was written by the other side. So — any position exceeding the buffer fails immediately. Extension walking does not recurse and bounds its steps by the buffer size (a run of zero-length extensions never ends an unbounded loop). Mismatched lengths are not read charitably — `ch_ok` requires the declared length to equal the buffer **exactly**, and `ch_ext_len` requires extensions to fit to the end.

op	What it does
`st_start` · `st_recvd_ch` · `st_negotiated` · `st_wait_flight2` · `st_wait_finished` · `st_connected`	State numbers
`hs_client_hello` · `hs_server_hello` · `hs_encrypted_extensions` · `hs_certificate` · `hs_certificate_verify` · `hs_finished`	Message type numbers
`next_ok` · `step`	May this message be received now · move state (yields the same state if it cannot)
`ch_ok`	Do type and length match the buffer
`ch_random_off` · `ch_sid_off` · `ch_sid_len`	Random · session_id
`ch_suites_off` · `ch_suites_len` · `ch_has_suite`	Offered suites
`ch_ext_off` · `ch_ext_len` · `ch_ext_find` · `ch_ext_find_len`	Extension block · find by type
`ch_x25519_off` · `be16`	Position of the x25519 public key in key_share · big-endian 2 bytes
`build_sh` · `build_ee` · `build_cert`	Build ServerHello · EncryptedExtensions · Certificate (one DER)
`cv_content` · `build_cv` · `build_fin`	The 130 bytes CertificateVerify signs · build CertificateVerify · Finished

`server_finished`	From ECDHE up the key schedule to the server Finished in one line
`app_secrets` · `check_client_finished`	Application traffic secrets (c · s) · check the peer's Finished (1 = match)
`traffic_keys` · `seal_app` · `open_app`	Secret → key ‖ IV (one direction at a time) · seal · open application data records

Table 50.1 — Ops of `tlssrv`

Message building shares one frame (`<type 1> <length 3> <body>`) — so the frame is written once. Repeating the same arithmetic in four places brings the day only one is fixed.

`cv_content` — the 64 spaces are not decoration. The signed content is `0x20 × 64 ‖ "TLS 1.3, server CertificateVerify" ‖ 0x00 ‖ transcript hash` (§4.4.3). Without that prefix, this signature could be reused as a signature in another context (a certificate signature, a client-side signature). Constants in a standard that make you ask "why is this here" are usually traces of attacks that already happened.

Order is half the standard. `next_ok` enumerates what is accepted, and enumerating is rejecting everything else. Without it a man in the middle could send Finished early or ClientHello twice. **`session_id` is echoed back** — TLS 1.3 does not use that field, but not echoing it gets connections cut by middleboxes on real networks, because it is a compatibility device to look like 1.2 (§4.1.3).

The transcript is cut three times in this layer. `s hs traffic` stands on `CH‖SH`, the server Finished on `CH‖…‖CertificateVerify`, and application traffic secrets and **the client Finished** on `CH‖…‖server Finished`. The client Finished does not include itself, and its verify_data comes from the **client** handshake secret — same transcript, different secret. Using the server's always rejects, and the symptom looks like "the client is broken".

Application data uses a different secret and a different sequence per direction. Sharing one makes nonces overlap, and overlapping nonces in AEAD lose plaintext and authentication key together. So key bundles are made one direction at a time. Sequences increase **per record** and restart at 0 per key generation. The output buffer of `traffic_keys` must be 32 bytes even when the key is 16 — `expand_label` uses one HMAC block and always writes 32. Alerts are the same records too — the inner type is 21, and without looking at it an alert is read as data.

What is checked — the real handshake of RFC 8448 §3. The canonical ClientHello's fields are read, and five messages are built and compared byte for byte with the canonical ones (ServerHello 90 · EncryptedExtensions 40 · Certificate 445 · CertificateVerify 136 · Finished 36). And it is woven end to end — we compute ECDHE from the canonical keys, climb the key schedule and match the server Finished with the canonical one. That one move measures the transcript cut points, the key schedule, finished_key and verify_data together. The canonical CertificateVerify is RSA-PSS and our signer is ECDSA, so the frame is what is measured here; the signature itself is measured separately by `ecdsa`.

`tlscli` — the TLS 1.3 client handshake

Source	`lib/tlscli.low`
Layer	L0 — pure computation (the caller's backing)
Capabilities	none

The **mirror image** of `tlssrv`. The server side reads a ClientHello and builds a ServerHello. This module does the opposite: it **builds** a ClientHello and **reads** the ServerHello and what follows. The protocol's computation (key schedule, record sealing) is in `tls13` this module is the layer that calls it in the client's order.

> **This alone is not a TLS client**
>
> **No transport** — no sockets, no reassembly; the caller gathers the bytes. That is why this module is `effects none`. **It does not check certificates** — it only takes certificates out of the `Certificate` message; the chain, validity and names are checked by the caller with `x509` · `verify` · `trust`. A tool that does not check must say so (`lowget`'s `--insecure`). Not built: HelloRetryRequest · PSK/0-RTT · session resumption · client certificates · key-exchange groups other than x25519 · suites other than `TLS_CHACHA20_POLY1305_SHA256` · `TLS_AES_128_GCM_SHA256`.

The order of the handshake

A client accepts messages in one order only. If that order is not kept, a man in the middle can drop or swap messages. `cnext_ok` lists **the one message that may arrive** in each state and refuses everything else.

```
 cst_start ──sends ClientHello──► cst_wait_sh
 cst_wait_sh ──ServerHello(2)──► cst_wait_ee        ── the handshake keys are made here
(hs_secrets)
 cst_wait_ee ──EncryptedExtensions(8)──► cst_wait_cert
 cst_wait_cert ──Certificate(11)──► cst_wait_cv     ── taken out with cert_at; the caller checks
them
 cst_wait_cv ──CertificateVerify(15)──► cst_wait_finished
 cst_wait_finished ──Finished(20)──► cst_connected  ── check_server_fin must be true
```

The number in parentheses is the handshake message type. `cstep` gives the next state.

ops

op	what it does
`cst_start` … `cst_connected`	state numbers (0 … 6)
`cnext_ok` · `cstep`	may this message arrive in this state · the next state
`build_ch`	builds a ClientHello — suites ChaCha20-Poly1305 first, then AES-128-GCM; key exchange x25519

`sh_ok` · `sh_usable`	is the ServerHello well formed · can we continue with it (decided in one call)
`sh_is_hrr`	is it a HelloRetryRequest — recognised and **refused**
`sh_suite` · `sh_is_tls13` · `sh_key_share_off`	the chosen suite · is it really 1.3 · where the server's x25519 value is
`sh_ext_off` · `sh_ext_len` · `sh_ext_find` · `sh_ext_find_len`	the extension block, and finding one extension
`hs_secrets`	makes the handshake secrets (the key ladder)
`finished_vd`	the Finished verify value from any traffic secret
`check_server_fin`	checks the server's Finished — "does the other side really hold the secret"
`build_client_fin`	builds our Finished
`seal_rec`	seals one record (the caller picks the inner type)
`plain_hdr` · `rec_len` · `rec_type`	builds a plaintext record header · reads a received record header
`cert_at` · `cert_len`	offset and length of the `n`-th certificate in a Certificate message (0 is the leaf)

Table 50.1 — ops of `tlscli`

Design

Every length on the reading side was written by the other side. Everything from the ServerHello on is bytes nothing has authenticated yet. Any offset past the buffer answers 0 at once, and a length that does not fit is not read "generously".

A HelloRetryRequest is not read as a ServerHello. An HRR has the same message type as a ServerHello and differs only by a fixed 32-byte value in the random field. Without knowing that, the HRR's random is taken as real and the key ladder is built on a **silently wrong** value. HRR is not built, so it is recognised and refused.

The extension picks the version, not the header. The `0303` in the message header is decoration to fool middleboxes. Whether it is really 1.3 is decided by the `supported_versions` extension being `0304` (`sh_is_tls13`).

Why ChaCha20 is offered first. The two ciphers written in this language were measured and the faster one goes first. A server that honours the order picks the faster one.

`check_server_fin` being true does not yet say who. It confirms "the other side holds the handshake secret"; who that side is, is the certificate's question (`verify`). Both must be checked before the connection can be trusted.

Why this module exists. With a client, a real outside implementation (`openssl s_server`, a real web server) can stand on the other side. Testing only against our own server (`tlssrv`) lets both

sides share the same misreading, and then “we read the spec the same way” is not confirmed. The real tool built on this module is `apps/lowget`.

`http` — HTTP/1.1 request parser

Source	`lib/http.low`
Layer	L0 — pure computation
Capabilities	none

Takes request bytes and answers **where each field starts and how many bytes it is** (RFC 9112). It does not cut — the caller cuts with `subslice`. **Every failure is `0`.** `0` can never be a legitimate position — the smallest request is `GET / HTTP/1.1␍␊␍␊`, so no field starts at 0.

What it promises and what it does not

Server-side request parsing only (no response parsing). There is no transport — the caller gathers the bytes. **Chunked transfer (`Transfer-Encoding: chunked`) is not built** — it only finds that header, so the caller must reject. No trailers, multi-line headers (obs-fold), URL percent-decoding or HTTP/2. HTTP/1.0 is not accepted — its persistent connection rules differ and were not built.

```
let t u64 be http.target_off b .
let n u64 be http.target_len b .
guard gt n 0 . else return 0 .
let target slice u8 . be subslice b t (add t n) .
```

op	What it does
`method_get` · `method_head` · `method_post` · `method_put` · `method_delete`	Method codes (1 … 5)
`method_code`	Method in the request line → code. 0 = unknown
`line_next` · `line_len`	Position of the next line · content length of this line (without CRLF)
`target_off` · `target_len`	Position · length of the request target
`version_ok`	Is it `HTTP/1.1`
`headers_off` · `header_next`	First header · next header (0 at the blank line)
`name_len` · `value_off` · `value_len`	Header name length · value position · value length
`name_eq`	Is this header's name that one (case-insensitive)
`header_find` · `header_find_len`	Value position · length for that name. **0 if duplicated**
`content_length`	`option u64` — `some 0` if absent, `none` if malformed
`body_off`	Where the body starts

Table 50.1 — Ops of `http`

The core of this module is rejection. A parser is defined more by what it rejects than what it accepts. Wrong acceptance in HTTP has a name — **request smuggling**. If the front (proxy) and back (server) read the same bytes differently, a request one sees the other does not.

Rejected	Why
Bare `LF` as line end	If the front accepts only CRLF, boundaries diverge
Space between name and colon (`Host : x`)	RFC 9112 §5.1 requires rejection
Two `Content-Length`	Rejected even if the values agree
`Content-Length: 5, 5` · `+5` · empty	Digits only — that leniency is smuggling
`HTTP/1.0`	Different persistent connection rules, not built
The same header twice (`header_find`)	Whether merging is allowed varies per header and that table was not built — unknown means reject
Empty target (`GET  HTTP/1.1`) · unterminated headers (no blank line)	—

Table 50.2 — What `http` rejects

Why `content_length` is an `option`. "Absent" and "wrong" are different answers. Absent gives `some 0` (a normal request without a body); malformed gives `none` (the connection must be dropped). With one value the two mix, and where they mix is where attacks live.

This module caught a processor defect. It was first written with `input b str ..` `--check` passed, but 16 of 21 ops fell onto the slow interpreted path (about 80×). `str` was not a builtin but a local alias in `strings` the checker let the name through, but the typed lowering did not know its meaning. The answers were right, so tests could never see it — a silent 80×. Now using such a name in a signature gives a `W-NOT-YET` warning. Passing checks and being fast are different things, and without looking at `--why-slow` this file would have shipped as it was.

What is checked — RFC 9112 examples (8 normal), 9 rejections, VM/native agreement, zero ops falling onto the slow path.

soa — an SoA layout trial: one array per field

Source	`lib/soa.low`
Layer	L0 — pure computation
Capabilities	none

There are two ways to keep n elements of `{x, y, vx, vy}`. **AoS** (Array of Structs) lays whole elements in one array; **SoA** (Struct of Arrays) keeps **one array per field** side by side. For a computation scanning only x, SoA reads just the needed values contiguously and uses the cache economically.

A measurement, not an answer

A study proposing SoA as a language feature (a type constructor like `store[T, soa]` with dozens of keywords and ops) was **deferred**, with the reconsideration condition "first write it in `lib/` with zero new builtins; where it gets stuck becomes the list of language work". This module is that. Finding — **the SoA layout itself is fully expressible as a library**. Per-field arrays, stride-free sequential access and single-field kernels are all written without new language machinery. No speed claims without measurement — this module guarantees only correctness, that both layouts give the same answer, and code touching whole elements is better off with AoS.

op	Shape	Failure
step_x	`proc (xs mut slice u64, vxs slice u64, n u64) → u64` — `xs[i] += vxs[i]`	none — processes only `min(n, len xs, len vxs)` and answers that count
sum_field	`fn (f slice u64) → u64`	none
get_x	`fn (xs slice u64, i u64) → u64`	`0` out of range
step_all	`proc (xs, ys mut slice u64, vxs, vys slice u64, n u64) → u64`	none — the smaller of two counts
step_x_aos	`proc (rows mut slice u64, stride, xoff, voff, n u64) → u64` — the AoS version	stops at the element whose velocity slot is out of range and answers that `i`

Table 50.1 — Ops of `soa` — all `effects none`

```
proc demo input xs mut slice u64 . . input vxs mut slice u64 . .
  input rows mut slice u64 . . output u64 .
do
  set (index xs 0) 1 .
  set (index xs 1) 2 .
  set (index vxs 0) 10 .
  set (index vxs 1) 20 .
  let n1 u64 be soa.step_x xs vxs 2 .
  guard eq n1 2 . else return 90 .
  let s1 u64 be soa.sum_field (subslice xs 0 2) .
  set (index rows 0) 1 .
  set (index rows 1) 10 .
  set (index rows 2) 2 .
  set (index rows 3) 20 .
  let n2 u64 be soa.step_x_aos rows 2 0 1 2 .
  guard eq n2 2 . else return 91 .
  var s2 u64 be add (index rows 0) (index rows 2) .
  guard eq s1 s2 . else return 92 .
```

```
  return s1 .
end
```

Where it gets stuck — the list of language work. ① There is no syntax for handling one element "as a lump" — the caller gathers fields by hand (`get_x`). Inconvenient, not impossible. ② Arguments multiply with the number of fields (`step_all`) — solved since slices were allowed in struct fields, but this module is a measurement record and keeps the four-argument shape. ③ **Types do not know the layout** — AoS and SoA versions get different op names (`step_x` versus `step_x_aos`), and AoS offset arguments are all `u64` where the compiler cannot help.

> **Counter-example. Ignoring the returned count**
>
> `soa.step_x xs vxs 1000` silently processes only 3 if `xs` has 3 slots. Code assuming all `n` were processed confirms with `guard eq m n ..`

> **Counter-example. Swapping the AoS offsets**
>
> `soa.step_x_aos rows 2 1 0 3` has xoff and voff reversed and adds position to velocity without error. That is how "layout is not in the type" feels.

Cautions. Keeping parallel arrays the same length is the caller's responsibility — ops shrink to the shorter one without telling. `get_x`'s failure value 0 cannot be told from a normal value. A bare `index` does not shrink the range and stops with `E-VM-BOUNDS`. Splitting a long `let` or `set` across lines lets the newline close the form — end the line with `,` to continue.

`allocs` — allocator traits, bumps and default allocators

Source	`lib/alloc.low`
Layer	L1 — storage
Capabilities	none for bumps · `fixed_bytes` spawns only where `cap allocator` is held · `heap_bytes` only where `cap heap` is held

The allocator **interface** (two traits) and four actors satisfying it. The most common use is a bump that **cuts pieces from the front** of a large borrowed byte block — suited to data "made in many pieces, discarded all at once" (a parser's temporary nodes, strings living for one pass) (chapter 35).

```
var bump allocs.bump_bytes be spawn actor allocs.bump_bytes . .
var c u64 be send bump init mem .
let b option mut slice u8 . be send bump reserve 64 .
```

The root and what sits on it. Lowent has no implicit global heap. The only place bytes first enter a program is the root op `alloc_bytes <capability> capacity n`, and there are two roots — a **fixed window** (`cap allocator`, effect `alloc`; on bare metal the linker sets the window's bounds) and a **heap** (`cap heap`, effect `heap`; only on hosted systems with an operating system) (chapter 20). `allocs` handles what sits **on** the root. It does not ask whether the bytes came from a root or were lent by a caller. So code without `cap allocator` still allocates fully on bytes someone else gave — capabilities divide abilities. It is written with actors, traits, options and subslices only, so it is a library, not a builtin.

```
export trait byte_allocator do
  reserve input s self . input n u64 . output option mut slice u8 . . effects state via self .
  grow    input s self . input old mut slice u8 . . input newn u64 . output option mut slice
u8 . . effects state via self .
  used    input s self . output u64 . effects state .
end
```

`freeing_allocator` adds `release input s self . input v mut slice u8 . . output bool .` to the same three. `via self` means "this op's effect is the effect the implementation declares" — a bump's `reserve` is just `state`, while `heap_bytes`'s `reserve` is `heap state`. So `heap` appears in the signature of container ops monomorphised with that allocator.

actor	Backing · policy	`reserve` effect · where usable
`bump_bytes`	buffer attached with `init` · advances the cursor by the request · both traits	`state` · anywhere
`bump_aligned`	same backing · rounds the start up to a multiple of 8 before cutting (padding is lost)	`state` · anywhere
`fixed_bytes`	capability slot `root cap allocator` · carves straight from the root	`alloc state` · anywhere (the linker window on bare metal)
`heap_bytes`	capability slot `root cap heap`	`heap state` · hosted only (`E-HEAP-NOHOST` on bare metal)

Table 50.1 — Actors of `allocs`

op	Shape	Failure

`init` (outside the trait)	`backing mut slice u8 → u64` (capacity), `effects state`	none. Calling again switches to the new buffer with the cursor at 0
reserve	`n u64 → option mut slice u8`	`none` — the cursor does not move (no partial allocation)
grow	`old mut slice u8, newn u64 → option mut slice u8`	`none` — not the last piece, or no room
release	`v mut slice u8 → bool`	`false` — changes nothing unless it is the last piece
used	`→ u64` (cursor position, including padding), `effects none`	none

Table 50.2 — Ops of the bumps

- **OOM is a value.** `reserve` returns `none` instead of stopping. Bytes cannot be reached without `guard is_some …`.
- **What is returned is a view, not an offset.** Writing into `reserve`'s result changes the backing buffer.
- **`grow` extends the last piece in place.** The test is "is the piece passed **exactly the bytes** last handed out" (`same_slice`). It used to check only length, and another buffer of the same length passed, making two containers silently overlap (found in a security review). **Failure is a value, and this is an optimisation, not a contract** — a new allocator is complete with `grow` as the one line `return none .`. This brought a growing vector's arena high-water mark from about 4× the request down to 2× (`growvec`).
- **`release` only accepts the last piece** — silently accepting an unknown piece would let a double release wipe someone else's place.
- `fixed_bytes` and `heap_bytes` have no `init`, and `grow` is always `none` (the root does not know whose the last allocation was). A capability slot is not a runtime value — the op spawning that actor must hold the same kind of capability (`E-CAP-FORGE`). A heap cannot be conjured in one line where no capability exists.

Code taking an allocator accepts any implementation with `input comptime a type . + using al a . + requires allocs.byte_allocator a .`. Callers pass it not as a positional argument but with `let x … using <source> be …`, and if the op has only one source it is the default without being written. Monomorphisation means no vtables and no indirect calls.

```
proc two_from .
  input comptime a type .
  using al a .
  output u64 .
  effects state via a .
  requires allocs.byte_allocator a .
do
  let p option mut slice u8 . . be send al reserve 3 .
  guard is_some p . else return 91 .
  let q option mut slice u8 . . be send al reserve 5 .
  guard is_some q . else return 92 .
  let g option mut slice u8 . . be send al grow (some_value q) 9 .
  return send al used .
end
```

```
proc borrowed2 input buf mut slice u8 . . output u64 . effects state . do
  var b allocs.bump_bytes be spawn actor allocs.bump_bytes . .
  let c u64 be send b init buf .
  let n u64 using b be two_from .
  return n .
end
```

With a bump, 3 + 5 grown to 9 gives 12; with `bump_aligned` the second piece starts at 8, giving 17; with `heap_bytes`, `grow` is `none` and `heap` appears in the instance's signature.

> **Counter-example. Unwrapping without checking `none`**
>
> `some_value (send a reserve 99)` stops with `E-VM-NONE` when the buffer is small. It compiles, so it is easy to be careless.

> **Counter-example. Reaching the root without capability · sending from a fn**
>
> An op without a root capability cannot call `alloc_bytes` (`E-ALLOC-NOCAP`, `E-HEAP-NOCAP` for the heap) — `allocs` is not a substitute. Calling handlers from an `effects none fn` is `E-EFFECT-CALC`. Passing a source that does not satisfy the trait is `E-BOUND-UNSAT`.

Cautions. Nothing but the last piece can be freed — for take-and-release patterns, `pool` fits. Scope-based bulk release belongs to `region` blocks (chapter 18). Views from `reserve` alias the backing buffer and do not vanish when `init` is called again. Actors assume sequential delivery — passing a bump to a task is rejected (`E-ALLOC-SHARED`: an allocator handed to a task must have an `atomic reserve`). The floor starts at 0 and no addresses are exposed, so VM and native see the same bytes.

`pool` — generational-handle block pool

Source	`lib/pool.low`
Layer	L1 — the caller's storage
Capabilities	none

A pool lending and taking back fixed-size blocks. Handles carry a **generation number** so stale handles are recognised. Used when objects are created and deleted repeatedly and you want to stop the defect of reusing something already deleted (use-after-free). "Take 4 KiB at a time as needed, free a particular block when you choose" has a release time that is not lexical, so neither `region` nor a bump will do — so instead of static checking, generational handles (chapter 18, chapter 35).

```
newtype pa u8 .
let po option (pool.block_pool pa) . be pool.init pa mem gens 4096 .
guard is_some po . else return 1 .
var p pool.block_pool pa . be some_value po .
let h option (pool.handle pa) . be pool.take pa p .
```

What it prevents and what it does not

Prevents ① — accessing a returned block through an old handle. The generation differs, so it is rejected (as a runtime value). **Prevents ② — mixing pools**. Handles and pools carry a **brand** as a type, so putting `handle pa` into `block_pool pb` is the compile error `E-TYPE-INSTANCE`. And `init` seals `mem` and `gens`, so ops no longer take them — the path to passing the wrong array vanished from the surface. A brand is a type, not a value, so handles do not grow. **Does not prevent** — one brand per declaration. Calling an op that takes a brand as a comptime argument and runs `init` twice makes one brand cover two pools. Keeping one brand per pool is your job. These are not generational handles built into the language — this is an ordinary library, and the safety comes from its discipline.

Principle. Each block has a generation number, copied into the handle on take. Releasing bumps the block's generation, and from that moment old handles no longer match and become invalid automatically. **Fixed size** makes release just pushing onto a free list, with zero fragmentation. Generations live in a parallel array `gens` (SoA), and the check cost falls only on code passing through the gate `bytes`. **Free list links live inside the blocks' own bytes** — a released block has a bumped generation and nobody can reach it, so using its first 8 bytes as the ledger is free. The pool is a struct, not an actor (actor state cannot hold slices).

op	Shape	Failure
`handle b · block_pool b`	structs — a handle has `blk · len · gen`; a pool has sealed `mem · gens` and cursors	—
`init`	`comptime b, mem mut slice u8, g mut slice u64, bs u64 → option (block_pool b)`	none if `bs < 8`
`blocks`	`fn (comptime b, p) → u64` — `min(len mem / bs, len g)`	none

take	`(comptime b, p mut block_pool b) → option (handle b)`	`none` if no blocks remain
release	`(comptime b, p mut, h handle b) → bool`	`false` for stale or out-of-range handles (double release included)
bytes	`(comptime b, p, h) → option mut slice u8` — the only gate to the bytes	`none` if stale or out of range
alive	`(comptime b, p, h) → bool`	none (false is the answer)
used · outstanding	`→ u64` — blocks ever handed out sequentially · blocks out right now	none

Table 50.1 — Ops of `pool` — the first argument is the brand, all `effects none`

`take` **gives back released blocks first** (the free list is LIFO) — repeated take and release never dries the pool. A reused block's handle is a new handle with a bumped generation. Why the brand is written every time — this language has no inferred type parameters, and that one word is the contract "this handle belongs to that pool".

Take the view in the borrow head — the slice `bytes` returns is a plain slice and knows nothing about release. So take the view in the **head of a borrow** — `borrow v be some_value (pool.bytes <brand> p h) do … end` — touch bytes only inside it, and release **after** the block (chapter 12). Taken that way, the processor rejects handing that pool to a writing position (`release`, `take`) or opening a second borrow from the same pool while the borrow lives: `E-BORROW-EXCL`.

```
newtype demo_brand u8 .

proc demo input mem mut slice u8 . . input gens mut slice u64 . . output u64 . effects none .
do
  let po option (pool.block_pool demo_brand) . be pool.init demo_brand mem gens 16 .
  guard is_some po . else return 89 .
  var p pool.block_pool demo_brand . be some_value po .
  let h option (pool.handle demo_brand) . be pool.take demo_brand p .
  guard is_some h . else return 91 .
  let hh pool.handle demo_brand . be some_value h .
  guard pool.alive demo_brand p hh . else return 92 .
  var total u64 be 0 .
  borrow v be some_value (pool.bytes demo_brand p hh) do
    set (index v 8) 3 .
    set (index v 9) 4 .
    set total (add (narrow u64 (index v 8)) (narrow u64 (index v 9))) .
  end
  let rel bool be pool.release demo_brand p hh .
  guard eq rel true . else return 94 .
  let dead option mut slice u8 . . be pool.bytes demo_brand p hh .
  guard eq (is_some dead) false . else return 95 .
  return total .
end
```

> **Counter-example. Reaching through a released handle · double release**
>
> After release `bytes` is `none`, and an unchecked `some_value` stops with `E-VM-NONE` on that line. A second `release` just returns `false` silently, so without looking at the return value a "believed released but not" defect hides. Receive it with `guard eq rel true ..`

Counter-example. Releasing with values that must survive in the first 8 bytes

Release overwrites that place with a free list link. It is safe because the generation is bumped and nobody can reach it, but the expectation "it stays in memory after release" is wrong for the first 8 bytes. Put data from byte 8 on.

Counter-example. Sending a borrowed name out of the block

`borrow v be bv do set out v . end` is the compile error `E-BORROW-ESCAPE` — a borrow ends at the end of its block.

Counter-example. Releasing inside the borrow

`borrow v be some_value (pool.bytes b p h) do pool.release b p h . … end` is the compile error `E-BORROW-EXCL` — a release changes the pool, and the borrowed view cannot know. Release after closing the block.

Counter-example. Using an old view after the release

`pool.release` declares `invalidates p .`, so a view bound with `let bv … be some_value (pool.bytes b p h) .` and used **after** the release is the compile error `E-VIEW-INVALIDATED` — the processor follows `bv` back to `p`. To reach the block again, ask `bytes` anew (the generation answers `none`).

Counter-example. Holding two write views of one handle outside a borrow

Calling `bytes` twice with the same arguments and binding two names gives one block two writers. Taken in borrow heads, the second borrow is rejected with `E-BORROW-EXCL`; two bound outside a borrow and both used are rejected with `E-EXCL`. Take one view, in the borrow head.

Cautions. Handles are values and can be copied, but releasing through any copy stales them all. `outstanding` not growing no matter how often you take and release is proof the free list is alive (`used` grows only in the first few rounds). `outstanding` walks the free list, O(free blocks) — do not call it on every hot-path pass. One handle = one block. Sequential delivery is assumed. Why `outstanding`, not `live` — the processor could not tell common words used as local variables from op heads. Designing a handle's bit widths is helped by `budget`.

`shard` — access units that split a store

Source	`lib/shard.low`
Layer	L1 — the caller's storage
Capabilities	none

Splits one block of storage into **non-overlapping pieces** so each can be changed separately. A piece is represented by a **token**, and a token is the right "this range is mine". Whether pieces then ride on several threads or run sequentially, the fact that they do not overlap is itself the value.

```
newtype grid u8 .
var r owned shard.token grid . be shard.open grid 8 .
var h shard.halves grid . be shard.split_at grid r 4 .
var lo owned shard.token grid . be (field h low) .
var hi owned shard.token grid . be (field h high) .
rem r can no longer be used --- using it is E-OWN-MOVED
```

What it prevents — and who prevents it

① Touching through the root after splitting is the compile error `E-OWN-MOVED`. The language prevents it, not this module — the token is `owned`, so it leaves your hands the moment it is passed to `split_at`. ② Touching another piece's positions makes `write` and `read` answer `false` and `none` (no stop — failure is a value). ③ Tokens of different stores have different brands as types and do not mix. **Not prevented** — viewing the other piece read-only while split (frozen cross reads) and region-level disjointness proofs. They were not built, and that is written down.

Why it was built without a new statement. The language design documents proposed a new statement, `split region R into R1 … Rn by P`, as the answer. But the language already had two things — **brands** (types carry a store's identity) and **`owned`** (a value is unique and leaves your hands when passed). Multiply them and a token is an access unit (chapter 19).

op	Shape	On failure
`token` · `halves`	Piece token · result of splitting in two (`low` · `high`)	—
`open`	`comptime b, n u64 → owned token b`	none (range `0..n`)
`split_at`	`comptime b, t owned token b, at u64 → halves b`	none (`at` is clamped to the range)
`rejoin`	`comptime b, a owned token b, c owned token b → option (token b)`	`none` if order or adjacency is wrong
`covers`	`comptime b, t token b, i u64 → bool`	—
`width`	`comptime b, t token b → u64`	contract violation stops on entry
`write`	`comptime b, t, mem mut slice u64, i, v → bool`	`false` out of range
`read`	`comptime b, t, mem slice u64, i → option u64`	`none` out of range

Table 50.1 — Ops of `shard`

`rejoin`'s ordering convention — and why it is not a contract. When joining, pass the lower id first (deadlock convention: cross acquisition is always ascending). Breaking it gives `none`. It was first written as `requires lt (field a id) (field c id) .`, but the contract test generator

cannot produce rejection cases for relations between two struct arguments. A contract nobody can verify is a sentence, not a check, so it was lowered to a value the caller must receive.

> **Counter-example. Joining in reverse**
>
> `shard.rejoin g hi lo` is `none`. It does not stop, so without looking at the return value you pass on unaware that nothing was joined. Receive it with `guard is_some back ..`

> **Counter-example. Opening two stores with one brand**
>
> Calling `shard.open` twice with the same brand `g` is the compile error `E-BRAND-REUSED`. A brand names one store. Two stores need two `newtype`s (brands carry no data, so they cost nothing).

Cautions. Bind tokens to `var` (`owned` requires a mutable place). `split_at`'s `at` is clamped — out of range yields an empty piece, and an empty piece touches nothing, so it is safe. There is no synchronisation on the hot path — the token already stated the right, leaving only a range check, and tests confirm zero atomic operations or locks on that path in emitted C. This module knows nothing of threads — putting tokens on execution units is the business of chapter 27.

budget — handle bit budgets and generation wraparound

Source	lib/budget.low
Layer	L1 — pure computation
Capabilities	none

Decides how many bits each of three pieces gets when a generational handle is packed into **one word**, and checks **at compile time** that the budget fits. It is the place for writing a handle's widths in source and having the tool enforce them when you build your own store — something that hands out and takes back slots, like `pool` or `shard`.

```
h = (generation << (shard_bits + slot_bits)) | (shard << slot_bits) | slot
      which life                    which shard               which slot
```

```
let ho option u64 . be budget.pack 32 8 24 7 1 0 .
guard is_some ho . else return 1 .
let h u64 be some_value ho .
let slot u64 be budget.slot_of 32 h .
```

What this module guards

A budget that does not fit does not compile. `budget.pack 40 16 16 …` sums to 72, more than a word — all constants, so it is `E-CONTRACT-IMPOSSIBLE` at the call site. **A value that does not fit its piece gives `none`** — truncating silently would make two different handles equal. **When a generation wraps, the slot retires** — `next_gen` answers `none` and does not revive it. Going back to 0 would revive every old handle for that slot, exactly what generational handles exist to prevent.

Machine	slot · shard · gen	Capacity	Grounds
64-bit	32 · 8 · 24	about 4.3 billion slots · 256 shards · about 17 million lives	the widest widely used real id (SQLite pages) is 32 bits
32-bit	16 · 4 · 12	65,536 slots · 16 shards · 4,096 lives	lwIP counts lengths in 16 bits

Table 50.1 — Recommended defaults and their grounds

These are only defaults — when a store writes its own numbers, those win.

op	Shape	On failure
`plan` · `default64` · `default32`	Budget struct · recommended defaults	—
`pow2`	`n u64 → u64` (contract `n < 63`)	contract violation stops on entry
`slots` · `shards` · `lives`	`p plan → option u64`	`none` if a width is 63 or more
`pack`	three budgets + three values → `option u64`	`none` if a value exceeds its piece · compile error if the budget overflows
`slot_of` · `shard_of` · `gen_of`	budget + `h` → `u64`	contract violation stops on entry
`retired`	`gen_bits, gen → bool`	—

next_gen	`gen_bits, gen → option u64`	`none` if the slot is retired

Table 50.2 — Ops of `budget`

> **Counter-example. Simply incrementing the generation**
>
> `let g u64 be add gen 1 .` gives no error, but past the width it equals a handle from another life. Use `next_gen` and accept `none` — it means that slot is finished.

> **Counter-example. Counting a 63-bit piece**
>
> `budget.pow2 63` is rejected by the entry contract. `shl 1 63` overflows signed 64 bits into a negative — rejecting beats silently returning a wrong number.

Cautions. Give **the same budget** when unpacking — if `pack` and `slot_of` see different numbers the answer is silently wrong. Write the budget once as three constants and use only those. Why the op is `pow2`, not `cap` — `cap` is a syntactic position (`input k cap clock .`), and as an op name it could not stand in one unit with `clock`. Two-word handles were not built.

`wire` — dividing one word into fields

Source	`lib/wire.low`
Layer	L1 — pure computation
Capabilities	none

When several values sit side by side inside one integer, this handles each field **with a single mask**. The mask says both where the field starts (position) and how many bits it has (width), so those two numbers, which always drift apart when written by hand, cannot drift. Carrying a date-time as one word (`clock.local_packed`), putting shard, slot and generation in a handle (`budget`), reading and writing device register fields — every place one word is used as several fields.

```
mask = 0x0f00        →  position 8 · width 4
wire.pick  mask w    →  (w & mask) >> 8       extract the field's value
wire.put   mask v    →  (v << 8) & mask       move a value into the field's position
wire.merge mask w v  →  (w & ~mask) | put     replace only that field of the word
```

```
fn m_month output u64 . do return 64424509440 . end
let mo u64 be wire.pick (m_month) w .
guard wire.fits (m_month) 9 . else return 1 .
let w2 u64 be wire.merge (m_month) w 9 .
```

What this module guards

> **Position and width come from one number** — "fixed the position but not the width" cannot happen. This module's tests keep the record of a hand-written complement mask off by 2^{32} that a test with zero arguments let through. **A zero mask is rejected** — every op carries `requires ne mask 0`. **The cost is zero** — `trailing_zeros` of a constant mask folds at compile time, and the machine instruction count equals hand-written code (x86-64 gcc -O2: extract 18 = 18, replace 27 = 27, measured by tests every run).

op	What it does
`shift_of` · `max_of` · `solid`	Mask position · largest value fitting the field · is the mask contiguous
`pick` · `put` · `merge`	Extract · move into position · replace only that field
`fits`	Does the value fit the field
`polarity v m`	Invert only the mask's bits (active-low)
`reverse_bits v w`	Reverse bit order within width `w`
`onwire order inv v w`	Apply a wire descriptor at once (compile-time values)
`permute v tab` · `is_identity tab`	Bit permutation · is it the identity
`to_set now want` · `to_clear now want`	Set mask · clear mask

Table 50.1 — Ops of `wire`

Byte order is the builtin `byte_swap`. The axes for going onto a wire (polarity, bit order, byte order, permutation) are independent of each other.

Why set/clear pairs, not "a value". Writing a whole value to a port means read → modify → write. If an interrupt turns on another bit in between, my write erases it. So hardware provides set-mask and clear-mask registers (STM32's `BSRR`) — nothing is read, so nothing is lost. This is

correctness, not convenience (chapter 30). **Identity is free** — `onwire 0 0 v w` has the same instruction count as code returning the value unchanged. A permutation table is a runtime value and does not fold, so `is_identity` lets you ask before calling.

> **Counter-example. Writing the position again separately from the mask**
>
> `bit_and (shr w 32) 15` has position (32) and width (15) as two numbers, so fixing one silently breaks. `wire.pick` takes one number.

> **Counter-example. Inserting without asking whether it fits**
>
> A month is four bits. `wire.merge (m_month) w 20` silently inserts 4 (= 20 & 15). `wire.fits` answers that question — the two are not one op because answering failure as a value would force unwrapping an `option` everywhere.

Cautions. Write each mask in one place (one name, like `fn m_month output u64 .`). Masks with holes (non-contiguous bits) are not handled — `solid` answers that. There is no width/frame axis (an 8-bit value in a 12-bit frame). Why the ops are not `get` or `read` — `get` is a word the processor already uses, so it would never be called, and `read` collided with C's `read` and broke native builds. Library op names must avoid the processor's words, the specification's tables and libc alike.

`flags` — named on/off settings in one word

Source	`lib/flags.low`
Layer	L1 — pure computation
Capabilities	none

Handles many on/off settings **in one integer**. Positions are called by **bit number** (0, 1, 2). Names live only at compile time; at run time one integer remains. Toggles on a settings screen, feature switches, dirty markers, device status bits — places where carrying each separately is wasteful and bundling makes comparison easy.

```
fn f_sound output u64 . do return 0 . end
fn f_music output u64 . do return 1 . end
let s0 u64 be flags.with (flags.empty) (f_sound) .
let playing bool be flags.has s0 (f_music) .
```

What this module guards

Nonexistent positions stop on entry — every op taking a position carries `requires lt n 64`. C's `1u << 64` is undefined behaviour; here it is rejection. **The cost is zero** — code written with the library and hand-written bit operations have the same instruction count (x86-64 gcc -O2: set 17 = 17, test 18 = 18, subset 24 = 24, measured by tests every run). **Bit numbers must not overlap** — the moment settings are saved to a file or device, the numbers become a promise. Deleting a middle position shifts what follows and silently changes the meaning of saved settings, so a deleted position stays empty and is never reused.

What it does	op
Empty value · one position	`empty` · `of n`
Set · clear · toggle · test	`with s n` · `without s n` · `toggle s n` · `has s n`
Union · intersection · difference · subset	`join a b` · `both a b` · `minus a b` · `inside a b`
How many · lowest position	`ones s` · `lowest s` (contract `s > 0`)
Any · none · all	`has_any s` · `has_none s` · `is_full s w` (width `w`) · `widthmask w`
What changed	`changes a b` · `changes_in a b m`
Only what differs from the default · replace only these positions	`from_default s d` · `overlay base over m`

Table 50.1 — Ops of `flags` — one word (up to 64 positions)

Beyond 64 positions it is a byte array with different names — `wide_bits s` (bytes × 8), `wide_has`, `wide_with`, `wide_without`, `wide_toggle`, `wide_ones`, `wide_next s from` (the next set position, skipping empty bytes whole). The cost at the boundary was measured — `has s 63` is 17 instructions, `wide_has s 64` is 26. The nine-instruction difference is the price of "which byte" (division, indexing) and "does that byte exist" (bounds check). That is why the two implementations are not hidden behind one name. Out of range is answered as a value (`false`, `0`) — the width is decided at run time, so it cannot be a contract.

Names are opt-in. The label table belongs to the caller; the library holds no strings. `flags.label_in "sound\x00music\x00shadow\x00" 2` gives `"shadow"`, and `label_count` answers the num-

ber of names. A program not using names has none of those characters in its binary — tests compare the read-only data size of both outputs.

When several flows modify the same word. This module's ops take a value and return a value — they do not modify a stored word. If several flows update with `set shared (flags.with shared 2)`, other updates vanish between read → modify → write. Atomic operations answer that — `atomic_or s 0 4` (set bit 2), `atomic_and s 0 251` (clear), `atomic_xor s 0 4` (toggle). With a constant position the mask is constant and it lowers to one machine instruction (chapter 27). For device registers use `to_set` and `to_clear` of `wire`.

> **Counter-example. Writing a position as a mask**
>
> `flags.with (flags.empty) 64` meant mask 64 (= position 6), but this module reads "position 64", and with both constant it is a compile error (`E-CONTRACT-IMPOSSIBLE`). Numbers here are always positions. To work with masks, use `wire`.

> **Counter-example. Saving settings only as their current value**
>
> When defaults change, saved values go stale. Recording only the difference with `flags.from_default s d` is small and keeps its meaning when defaults change.

Cautions. Fields wider than one bit are `wire`'s job. Bit numbers are not managed automatically — people write them, and a check in the development repository catches two names using the same position. Why the ops are not `set`, `union`, `count` or `first` — they are syntax words, existing operations, or were used as local names in other files. Exporting `first` broke `lib/regex.low`, and the combination check caught it.

segarena — fixed-size segment arena

Source	`lib/segarena.low`
Layer	L1 — the caller's storage
Capabilities	none

Grows storage not as one block but in **equal-size segments**. Inside a segment memory is contiguous, so scanning is fast, and growth **never moves what already exists**. One index splits into a segment and a slot — that is all.

```
segment = i >> kbits          slot = i & ((1 << kbits) - 1)
address = directory[segment] + slot
```

```
newtype grid u8 .
var a segarena.arena grid . be segarena.open grid 4 .
let b option u64 . be segarena.grow grid a dir 1024 .
guard is_some b . else return 1 .
let v option u64 . be segarena.at grid a dir mem 9 .
```

The cost is written in the op name — this is the point of the module

The same sum over 20.48 million elements was measured two ways. `sum_seg`, borrowing each segment once, looks at the directory once per segment and took 21 ms; `sum_global`, reading by global index every time, looks per element and took 81 ms. The answers are equal. The compiler does not know about segments — choosing the fast path is **the op you call**. The cost of global random access is not hidden as if it were a flat slice.

op	What it does	On failure
`arena`	The arena type (holds only a cursor and shape)	—
`open`	Opens an arena (`2^kbits` slots per segment)	contract `1 ≤ kbits ≤ 20` — rejected at compile time for constants
`grow`	Opens one more segment and records its start in `dir`	`none` if no room or over capacity
`seg_slots` · `seg_of` · `slot_of`	Slots per segment · global index → segment · slot	—
`at`	Read one slot by global index (convenient, consults the directory every time)	`none` if out of range or segment not opened
`walk_base`	Borrows a segment's start once (for hot traversal)	`none` if segment not opened
`sum_seg` · `sum_global`	The same sum by two paths — witnesses of the cost comparison	—

Table 50.1 — Ops of `segarena`

> **Counter-example. A meaningless segment size**
>
> `segarena.open g 0` is the compile error `E-CONTRACT-IMPOSSIBLE` — a one-slot segment is not a segment. The argument is constant, so it is decided at the call site (chapter 14).

> **Counter-example. Hot traversal by global index**
>
> Calling `segarena.at` per element inside a `while` gives no error, only slowness (3.9× in the measurement). Traverse segment by segment and borrow the start once with `walk_base`.

Cautions. Storage belongs to the caller — `dir` (start of each segment) and `mem` (where slots live) come from outside (the same discipline as `pool` and `shard`). Reclamation is bulk — there is no way to return one segment. If individual reclamation is needed, `pool` is the place. Variable-size segments are not built — the `i >> k` formula would not hold and the cost would change, so that must be a different module. Views borrowing segments belong to `segview`.

pagecache — page ids and pinning cursors

Source	`lib/pagecache.low`
Layer	L1 — the caller's storage
Capabilities	none

Separates **a long-lived name** (page id) from **short-lived access** (a pin) in a cache holding pages. What is stored outside is always the id; a pin is held only while touching bytes. SQLite's page cache has this shape. Pages move (eviction, relocation). When they move, the only thing to fix should be the id → slot table; a cache that requires hunting down scattered pointers is unusable.

```
newtype db u8 .
var c owned pagecache.cache db . be pagecache.open db 16 .
var p pagecache.pin db . be pagecache.acquire db c 42 .
rem evict cannot be called in between --- it does not compile
var c2 owned pagecache.cache db . be pagecache.release db p .
var c3 owned pagecache.cache db . be pagecache.evict db c2 .
```

Evicting while a pin lives is a compile error

`acquire` **swallows** the cache token. So while a pin lives there is nothing in hand to pass to `evict` or `reset` — `E-OWN-MOVED`. What prevents it is the language, not this module. `owned` already enforces "there is one token, and handing it over lets it go" (chapter 19). Brands seal a store, tokens divide access units (`shard`), and here the same rule makes a new protocol for free a third time.

What	Decided	Grounds
Default pin limit	`machine.cache_line / 8` — x86_64 and arm64 8 · mips_be 4 · cortex_m 1	PostgreSQL sized its pin array at 8 entries, "64 bytes, about the size of a cache line"
Release order	LIFO — a stacked pin swallows the previous one	SQLite cursors hold pages as a stack

Table 50.1 — Limits and order — where the numbers came from

The reason was written instead of the number. Hard-coding 8 is wrong on machines with a different cache line — so it divides `machine.cache_line` (chapter 30).

op	What it does	On failure
`cache` · `pin` · `pin2`	Cache token · pin · stacked pin types	—
`open`	Makes a cache token	—
`acquire`	Pins a page — swallows the cache token	—
`acquire_more`	Stacks a pin — swallows the previous pin (depth +1)	—
`release` · `release_more`	Releases — gives the token back	—

`evict · reset`	Evict · clear. Requires the cache token	compile error while pinned
`slot_of`	Current slot of an id	`none` for an empty cache
`pin_limit · within_limit`	This machine's limit and the check	—
`epoch_of · depth_of · depth2_of`	Observation	—

Table 50.2 — Ops of `pagecache`

> **Counter-example. Releasing stacked pins from the inside**
>
> Pin `a`, stack `b` with `acquire_more db a 9`, then call `release db a` — a compile error, because `a` is inside `b`. There is no way to break the order. That is LIFO.

> **Counter-example. Storing a slot for later**
>
> Holding a slot from `slot_of` and using it after an eviction gives no error, only a wrong slot. What is stored outside must be the id — slots are obtained afresh each time. That is why this module separates ids from slots.

Cautions. Store neither slots nor pins — both belong to this moment. The limit is a convention, not enforcement (depth is built only up to `pin2`). Reading pages in and its failure, and eviction policies like LRU, are not here — policy sits on top of this protocol.

`growvec` — self-growing byte vector

Source	`lib/growvec.low`
Layer	L1 — storage (carries an allocator)
Capabilities	none — `open` is handed a bump allocator

A **byte** array that fetches more room by itself when short. Used when you do not know how much will come — until the file is fully read, until the user presses Enter. `growvec.gvec` is an **alias** of `vecgen.vec u8 allocs.bump_bytes`, and its ops delegate to `vecgen`. For pushing bytes only, this one is shorter because no type arguments are written — a short name is interface too.

```
let g option growvec.gvec . using bump be growvec.open 16 .
guard is_some g . else return 1 .
var v growvec.gvec be some_value g .
guard growvec.add_all v "hello" . else return 2 .
guard growvec.add_all v " world" . else return 3 .
let s slice u8 be growvec.view_of v .
```

Why it exists. Gathering bytes with `vecs` means passing allocator, buffer and length on every call, and **rebinding** to the grown buffer is the call site's job too. Forgetting that one line is not caught by the compiler — the old buffer is still a valid slice. `growvec` puts the three **in one value** and removes that kind of defect structurally. There is still no magic — the allocator was still handed in by someone, and the cost of growth is visible at ops that can return `false`.

op	effects	What it does
`gvec`	—	alias of `vecgen.vec u8 allocs.bump_bytes`
`open cap0 (using al)`	state	Opens an empty vector. `cap0` may be 0 (becomes 8 on the first push). `none` if no place
`count_of g · cap_of g`	none	Bytes held · current capacity
`add_byte g b`	state	One byte — grows to double + 8 when short. `false` = no new place (the vector stays valid)
`add_all g s`	state	A whole byte string. **If it fails midway, the front part is already in**
`reserve_more g more`	state	Reserves room for `n + more` ahead
`view_of g`	none	A view over only what is held

Table 50.1 — Ops of `growvec`

`add_all` does not promise all-or-nothing. Promising it requires reserving room first, and whether to pay that is the caller's choice, so `reserve_more` is separate.

```
proc put_record input v mut growvec.gvec . input rec slice u8 . output bool . effects state . do
  guard growvec.reserve_more v (add (len rec) 1) . else return false .
  guard growvec.add_all v rec . else return false .
  return growvec.add_byte v 10 .
end
```

The memory side. A bump allocator never gives back, so once every growth left the old buffer abandoned in the arena, and holding 2,048 bytes used 8,104. Giving the allocator **in-place growth** (`grow` — if the last allocation is our buffer, extend it) removed those generations. The remaining factor of two is the growth policy's slack (capacity ≈ 2n), a normal trade against

time. If the size is known ahead, `reserve_more` removes even the slack and takes exactly what is needed (`allocs`).

> **Counter-example. Using a view taken before growth after growth**
>
> Taking `view_of` and then calling `add_all` may change the buffer, leaving the view on the old one. If it did not grow it is right by accident, which is worse. Take views last.

> **Counter-example. Copying a `gvec` and pushing to both**
>
> After `var b growvec.gvec be a .`, pushing to `a` and `b` separately makes both see the same buffer while each `n` is unaware of the other's change — they overwrite each other. Handle one vector by one name, and pass it as `mut gvec`.

> **Counter-example. Ignoring return values · reading up to `cap_of`**
>
> Ignoring `false` from `add_byte` lets the vector go on without that byte. Past `n` is garbage — use `count_of` or `view_of`.

Cautions. No individual release or shrinking. A vector lives within its allocator's lifetime. It is `effects state`, so the pure layer cannot use it — for problems solvable purely, caller buffers (the convention of `fmt`) are cheaper.

vecgen — generic self-growing vector `vec t a`

Source	`lib/vecgen.low`
Layer	L1 — storage (carries an allocator)
Capabilities	none — `open` is handed an allocator

An array holding any type that grows itself when short of room. It has two axes — **element type `t` × allocator type `a`**. One source yields instances like `vec#u32#allocs.bump_bytes` and `vec#u32#allocs.bump_aligned` (visible in `--ir`), with no vtable and no indirect calls — the policy sits outside the container, and choosing costs nothing (chapter 22, chapter 34).

```
let g option (vecgen.vec u32 allocs.bump_bytes) . using bump be vecgen.open u32 2 .
guard is_some g . else return 1 .
var v vecgen.vec u32 allocs.bump_bytes . be some_value g .
guard vecgen.append u32 allocs.bump_bytes v 100 . else return 2 .
guard vecgen.append u32 allocs.bump_bytes v 101 . else return 3 .
guard vecgen.append u32 allocs.bump_bytes v 102 . else return 4 .
let x option u32 . be vecgen.at u32 allocs.bump_bytes v 0 .
```

Types are written every time. Not for lack of inference — it was deliberately left out. The source shows which instances were created (how many copies of code end up in the binary), and readers know `v`'s element type without tracing back to its declaration. Inference can be added later; the reverse is hard. For bytes only, `growvec`, which takes no type arguments at all, is shorter.

Design. ① **Monomorphisation** — `vec u32` becomes a concrete struct at compile time, with every size and offset a constant. The price is code per instance, a visible cost, not a hidden one. ② **Storage is bytes** — because allocators hand out bytes. `view_array` presents them as elements, and `size_of t` answers how many bytes are needed. Without it, generic code would reserve the maximum width (8 bytes) and `vec u8` would use 8×. ③ **Rights come from elsewhere** — the container does not create rights, it only carries them.

op	effects	What it does
`vec`	—	struct — `al a · store mut slice u8 · n u64` (number of **elements** held)
`open t cap0 (using al)`	state via a	Opens an empty vector (room for `cap0` elements). `none` if the first place cannot be obtained
`count_of t a g · cap_of t a g`	none	Elements held · current capacity (in elements)
`append t a g x`	state via a	One element — grows to double + 8 elements when short. `false` = no new place
`at t a g i`	none	The i-th element → `option t`
`set_at t a g i x`	none	Changes the i-th. `false` out of range — does not silently grow
`reserve_more t a g n`	state via a	Reserves room for n more elements ahead
`view_of t a g`	none	A `slice t` over only what is held

Table 50.1 — Ops of `vecgen` — the first two arguments are element type and allocator type (`open` takes only the element type)

Units are not mixed — `cap0`, `cap_of`, `count_of`, `at` and `set_at` all count **elements**; bytes never leave the module. Elements are sized scalars or **viewable structs** (every field has a byte layout, 255 bytes total or less) — `at` returns such a struct with its fields intact. The result of `view_of` can go straight to `sortgen` or `searchlib`.

Reserving first gives all-or-nothing. After `reserve_more` succeeds, the following `append`s cannot fail midway.

```
guard vecgen.reserve_more u16 allocs.bump_bytes v (len src) . else return false .
var i u64 be 0 .
while lt i (len src) . do
  guard vecgen.append u16 allocs.bump_bytes v (index src i) . else return false .
  set i (add i 1) .
end
```

> **Counter-example. Using an old view after growth**
>
> Holding `s` from `view_of` and calling `append` may change the storage, leaving `s` looking at the old buffer. If it happened not to grow, it is right by accident, which is worse. Take views last.

> **Counter-example. Ignoring `append`'s `false` · reading up to `cap_of`**
>
> `false` is OOM; ignoring it leaves a silently shorter vector. Past the count is garbage — use `count_of` or `view_of`.

> **Counter-example. Structs without layout · nesting**
>
> Opening with a struct that has a slice field fails because there is no byte layout and so no array (`E-IR-UNDEF`). There is no nesting like `vec (vec u8 …)` either — a type argument is one word.

Cautions. No individual release or shrinking — a bump allocator only releases wholesale, so reduce growth with `cap0` or `reserve_more`. A vector lives within its allocator's lifetime. Owning elements (`vec files.handle`) currently fail inside the instance, and the diagnostic does not point at the call site — a remaining defect. The caller-held-buffer container is separate, `vecs` — the ownership model differs, so they were not folded together.

`mapgen` — generic hash map `table k v`

Source	`lib/mapgen.low`
Layer	L1 — storage (carries an allocator)
Capabilities	none — `open` is handed a bump allocator

A table finding values by key, with key and value types **of your choosing**. Each op call writes the key type and value type in that order (for the same reason as `vecgen`). `hashmap` is fixed to `u64 → u64`, keeps storage with the caller and makes you call `rehash` yourself. `mapgen` has open types, one value owns storage and allocator, and it rehashes itself when full.

```
let mo option (mapgen.table u32 u64) . using bump be mapgen.open u32 u64 4 .
guard is_some mo . else return 1 .
var m mapgen.table u32 u64 . be some_value mo .
guard mapgen.insert u32 u64 m 7 900 . else return 2 .
let x option u64 . be mapgen.lookup u32 u64 m 7 .
guard mapgen.erase u32 u64 m 7 . else return 3 .
```

Design. ⓪ Why `table`, not `map` — `map` is a builtin op. When one word means two things, how sentences group becomes unstable. Generics had been slipping past that check; closing the hole caught this module first. ① **No key value is reserved for empty** — a separate status array (`flags`: 0 empty · 1 occupied · 2 tombstone). The common shortcut "key 0 = empty" pretends key 0 is stored and silently gives wrong answers. ② **Tombstones remain** — turning a deleted slot back to empty makes keys placed beyond it unfindable forever. ③ **Load factor ≤ 0.5** — before passing half, slots double and everything is rehashed. ④ The hash is for distribution, not cryptography (one round of multiply-xorshift). ⑤ **Traversal is by slot, not iterator** — a hash table has no natural "next element", so instead of an iterator object carrying state, `next_used` gives the next occupied slot. There is no object to invalidate.

op	effects	What it does
`table`	—	struct — `al` · `keys` · `vals` · `flags` (bytes) · `n` (pairs held)
`open k v want (using al)`	state	Opens an empty map (ready for `want` pairs; slots are `(want+1)*2` rounded up to a power of two, at least 8)
`insert k v m key val`	state	Inserts (overwrites if present). Rehashes first if the load factor would be exceeded. `false` = no room
`lookup k v m key` · `has k v m key`	none	Value → `option v` · is it present
`erase k v m key`	none	Deletes (leaving a tombstone). `true` if it was present
`count_of k v m` · `slots_of k v m`	none	Pairs held (tombstones not counted) · slot count
`next_used k v m from`	none	First occupied slot from `from` → `option u64`
`key_at k v m at` · `val_at k v m at`	none	Key · value at that slot. `none` if not occupied

Table 50.1 — Ops of `mapgen` — the first two arguments are key type and value type

The key array is `slots × size_of k` bytes and the value array `slots × size_of v` bytes — opening `table u8 u16` for 3 pairs takes 32 bytes with 8 slots; the same count as `table u32 u64` takes 104. The arithmetic differs per instance.

```
var at u64 be 0 .
var total u64 be 0 .
var going bool be true .
```

```
while going . do
  let nx option u64 . be mapgen.next_used u32 u64 m at .
  guard is_some nx . else do
    set going false .
    continue .
  end
  let s u64 be some_value nx .
  let v option u64 . be mapgen.val_at u32 u64 m s .
  guard is_some v . else return none .
  set total (add total (some_value v)) .
  set at (add s 1) .
end
```

The frequency-count idiom is `lookup` → 1 if absent, +1 if present → `insert`. With value type `u8` and only 1 stored, it is a set.

> **Counter-example. Inserting while traversing**
>
> If `insert` rehashes, slot numbers change meaning and entries are skipped or seen twice. Collect what to insert and insert after traversal. `erase` is safe during traversal (it only leaves tombstones).

> **Counter-example. Not advancing `at` by one**
>
> `set at (some_value nx)` finds the same slot forever. The next start is **previous slot + 1**.

> **Counter-example. Non-integer keys · swapping type order**
>
> `table f32 u64` is unusable because `widen u64` produces meaningless bits — byte string keys are `strmap`. Opening `table u32 u64` and calling with `u64 u32` calls a different instance (the type checker catches it).

Cautions. Keys and values are integer scalars. `none` from `lookup` is not an error but the common normal case "that key is absent". `erase` decreases `count_of`, but slots do not shrink. Order is not promised (hash order, changing on rehash). Growth rehashes everything, costing O(slots) at that moment, and on a bump three old arrays remain — giving a proper `want` is the practical gain. A map lives within its allocator's lifetime.

`nodelist` — fixed intrusive lists

Source	`lib/nodelist.low`
Layer	L1 — the caller's storage
Capabilities	none

Nodes hold their links **inside themselves** (intrusive). No separate slot is allocated on the list side, so linking and unlinking allocate nothing — the shape of the Linux kernel's `list_head`. Only two things are dangerous — **moving** a linked node rots the places its neighbours pointed to, and **reclaiming** while linked rots the list.

The language prevents both. A node has a **move right** (`site`), and linking it into a list consumes that right (`held`). A program trying to move a linked node has nothing to hand over and is rejected at compile time.

```
newtype lru u32 .
var s owned nodelist.site lru . be nodelist.nl_open lru 3 .
var h owned nodelist.held lru . be nodelist.nl_link lru s .
rem nodelist.nl_relocate lru s 9  →  E-OWN-MOVED (s was already consumed)
var s3 owned nodelist.site lru . be nodelist.nl_unlink lru h .
var s4 owned nodelist.site lru . be nodelist.nl_relocate lru s3 1 .
```

op	What it does
`site` · `held`	Move right (unlinked) · linked ticket — branded types
`nl_open` · `nl_relocate`	Open a slot as a move right · move an unlinked node
`nl_link` · `nl_unlink`	Consume the move right, yield a linked ticket · consume the ticket, give the move right back
`nl_slot_of` · `nl_held_slot`	Slot number of a move right · of a linked ticket
`nl_solo` · `nl_after` · `nl_cut`	Make a lone ring · insert after `at` · remove (updating `nx` and `pv`)
`nl_walk_cut`	Walks from the head removing one node (mutable traversal)
`nl_count`	Number of nodes in the ring

Table 50.1 — Ops of `nodelist`

A list is a ring. With no null there is no value to mark the end. Instead, returning to the head is one full lap. A lone node is a ring pointing to itself. At first counting stopped at "points to itself", which was wrong — in a ring of two or more nobody points to itself, so that condition never comes.

When removing during traversal, read next first. Removing the current node makes its `next` point to itself, so without reading it beforehand the traversal stops right there. `nl_walk_cut` does this.

Storage belongs to the caller. The two slices `nx` (next) and `pv` (previous) come from outside. Links are **indices** — there are no raw pointers, and indices mesh with storage that does not move, like `segarena`.

Nodes in several lists at once. If the same node is in an LRU list and a hash bucket, there is only one move right, so the second link cannot happen. Split it with `shard` tokens — one piece per role, and the move right stands again only after all are rejoined (`rejoin`). That is exactly "unlink from every role before reclaiming". One brand names one store, so using the same brand for

node slots and role tokens is rejected with `E-BRAND-REUSED` — this actually happened while writing this manual, and the compiler was right.

Not built — automatic reclamation (the first reclamation policy is bulk), cycle detection, concurrent traversal, relocating linked nodes (fixing is the opposite).

`segview` — cursors, total length and coalescing for segment views

Source	`lib/segview.low`
Layer	L1 — the caller's storage
Capabilities	none

The language holds only three segment view operations — `view_segments` (build), `segs` (segment count), `seg` (the i-th segment). This module adds **cursors, total length and coalescing** on top. All three are written with those three, so there is no reason for them to be builtins.

Why `segments` is in the language. Without holding several segments as one value, sending them out takes one call per segment. Sending 64 KiB as 64 segments differed by 28× in the development repository's measurement (one `writev` versus 64 `write`s). A segment is an `iovec` — a slice in emitted C has the same size and both offsets as `struct iovec`, so passing a segment array across an `extern` boundary is a cast, not a copy (a test compares it every run).

```
rem the descriptor is (at, n) pairs in a flat slice u64 --- no new struct type
set (index d 0) 0 .  set (index d 1) 4 .
set (index d 2) 8 .  set (index d 3) 4 .
let ss be view_segments back d .
let n u64 be segview.total_len ss .
let w option u64 . be segview.coalesce ss out .
```

Call	What it touches	Cost
`segs` · `seg` (builtin)	descriptor only	a few ns or less per segment, zero copies
`segview.total_len`	descriptor only	O(segments)
`segview.cursor` · `seek` · `byte_at`	descriptor only	O(segments) — scans segments to find the position
`segview.coalesce`	**every byte**	O(n) — the only place copying happens

Table 50.1 — The cost is written in the op name

That is the point of the design — traversal only borrows segments, and the cost of flattening into one piece is paid only when called. The cost is visible at the call site.

Things to know. Storage belongs to the caller — `coalesce` writes into the given buffer and answers **the length written**; if the buffer is short it does not truncate but answers `none`. No cursor pointing to a nonexistent position is made — `seek` past the end is `none`.

One name hurt other modules. `total_len` was first called `total`. Then the library pair tests caught breakage when `segview` was used together with `tls13` or `tlssrv` — it collided with a local variable `total` inside `tls13` and produced an unrelated diagnostic (`E-GUARD-FALLTHROUGH`). Common words are not exported. (The diagnostic not reporting the name collision as a name collision is a separate remaining defect.)

Not built — variable element width (T of `segments T` is a scalar slice today), zero-copy optimisation of `coalesce`, the read-side (`readv`) counterpart.

`lifemode` · `lifeatom` — when a value ends

Source	`lib/lifemode.low · lib/lifeatom.low`
Layer	L1 — the caller's storage
Capabilities	`cap atomic` for `lifeatom`

There are four ways to decide when a value ends, and each has a different cost. No new word or builtin was added — this only bundles what the language already knew into something convenient.

Mode	Written with	Cost
① One owner	`owned` — one word does it (chapter 19)	no counting
② Thread-confined reference count	`lifemode.near_*` — plain addition	0 atomic instructions
③ Atomically shared reference count	`lifeatom.rc_*` + `cap atomic`	atomic instructions present
④ External completion	a completion token as an `owned` ticket	no counting; the compiler counts instead

Table 50.1 — Four ways to decide when a value ends

```
guard lifemode.near_open c 0 . else return 1 .
let a option u64 . be lifemode.near_share c 0 .
let b option u64 . be lifemode.near_drop c 0 .
rem answering 0 means it was the last --- the caller then reclaims the storage
```

The atomic side is a different module — `lifeatom.rc_open`, `rc_share k c`, `rc_release k c` (`k` = `cap atomic`). `rc_release` answering 0 means it was the last. ④ is not a library but a shape — make completion **consume** the token, as in `fn finish input t owned mymod.ticket . output result u64 mymod.late .`, and completing twice is `E-OWN-MOVED`, forgetting to complete (when completion can fail) is `E-OWN-INCOMPLETE`. **Exactly once** comes from the language, not a library.

op	What it does
`lifemode.ceiling`	The count ceiling both modes share
`lifemode.near_open` · `near_share` · `near_drop`	Confined count: open at 1 · add one · remove one (0 = last)
`lifeatom.rc_open` · `rc_share` · `rc_release`	Atomic count: the same three, taking `cap atomic`

Table 50.2 — Ops of `lifemode` · `lifeatom`

Why two modules — `use` costs what is exported, not what is called. At first all four were in one module. Then the binary of a program using only confined counting still had 2 `lock`-prefixed instructions. Exported ops get global symbols through C ABI wrappers, so the linker cannot discard them. Splitting the atomic side into `lifeatom` brought it to 0. The cost of bringing in atomics is visible in one line, `use lifeatom`.

Overflow — it saturates, and that object lives forever. Counting with plain atomic addition silently wraps to 0 at the ceiling (confirmed by running natively). A wrapped count reaches 0 early and frees a live object — the road to use-after-free. So this module **saturates** (the same

answer Linux `refcount_t` chose). At the ceiling it neither increments nor decrements. That object never dies, but memory safety is not broken. The cost is about 19 % more instructions; the atomic side's `lock` count stays at one per attempt (a CAS loop only spins under contention).

Not built — true asynchronous completion (no surface without backing), completion callbacks (they hide call timing and effects), weak references, cycle collection.

`io` — stream reading over slices

Source	`lib/io.low`
Layer	L2 — the pure half of the outside world
Capabilities	none — only the side filling the bytes needs capabilities

Reads bytes already in memory **like a stream**, a little at a time. Used to scan a fully read file line by line, or when a parser needs a cursor. The reading surface is not `read(buf) → n` but **peek · take · toss → views** — instead of filling caller memory, it hands out borrowed views pointing into the original (zero copies).

```
var r io.mem_reader be spawn actor io.mem_reader . .
let z u64 be send r attach src .
let line option slice u8 . be send r take_line 4096 .
```

Why no capability is needed. This module only cuts bytes already in memory and never reaches the kernel. Only the side actually filling bytes (`read_in` needs `cap io`, `files.read` needs `cap file_system`) requires a capability. So all framing logic (buffer management, `take_line`) is verified by VM/native comparison before touching the kernel, and kernel reading stays a thin leaf feeding bytes into that verified logic (chapter 28).

> **`none` is not failure**
>
> `none` from `peek`, `take`, `take_until` and `take_line` is **absence** — "the requested bytes are not there". `mem_reader` is pure arithmetic with nowhere to fail. So when `files` moved to the three-way `result (option T) file_error`, this module did not follow — adding `result` here would make every caller write an `error` branch that can never happen. The three-way answer is needed by a reader of raw fds, which does not exist yet.

op	Shape · effects	Failure
`scan_until`	`fn (src slice u8, from u64, delim u8, limit u64) → option u64`	`none` if not within limit or at the end
`mem_reader`	actor — state `src slice u8 · pos u64`	—
`attach s`	`→ u64` (always 0), state — attaches the source, cursor 0. Calling again restarts	none
`remaining`	`→ u64`, none	none
`peek n`	`→ option slice u8`, none — looks without consuming	`none` if fewer than n remain
`take n`	`→ option slice u8`, state	`none` if short, cursor unchanged
`take_rest`	`→ slice u8`, state	none (an empty view at the end)
`toss n`	`→ u64` (bytes actually dropped), state	none (drops only up to the end)
`take_until delim limit`	`→ option slice u8`, state — consumes the delimiter but excludes it from the view	`none` if not found, cursor unchanged

take_line limit	→ option slice u8, state — take_until 10 limit	none if no LF within limit

Table 50.1 — Ops of io

Design. EOF is none, not an error. **The limit is a required argument** — no default, so hostile input cannot trigger unbounded reads, and only the caller who knows the data knows how long a line may legitimately be. **No partial consumption** — on none the cursor does not move, so retrying with another length is safe. **LF is the only newline** — CR stays in the content. The library does not silently delete bytes (deletion cannot be undone; not deleting lets the caller choose). Positioning is the pure primitive scan_until, and the stateful cursor (actor) is a thin shell over it — to hold the cursor as a value (reentrancy, backtracking), use scan_until with a caller-held pos.

```
proc count_lines input src slice u8 . output u64 . effects state . do
  var r io.mem_reader be spawn actor io.mem_reader . .
  let z u64 be send r attach src .
  var lines u64 be 0 .
  var going bool be true .
  while going . do
    let l option slice u8 . be send r take_line 4096 .
    if is_some l . do set lines (add lines 1) . end
    if eq (is_some l) false . do set going false . end
  end
  if gt (send r remaining) 0 . do set lines (add lines 1) . end
  return lines .
end
```

count_lines knows neither files nor stdin — main fills bytes with one capability-holding line (read_in out 0 buf) and passes them via subslice. A final piece without a newline is not given by take_line, so remaining checks for it.

Counter-example. Treating take_line's none as EOF

none means (a) already at the end or (b) no LF within the limit. Not collecting the tail silently drops a final line without newline, and an unusually long line makes it look as if nothing is read from then on. Tell them apart with remaining.

Counter-example. Comparing CRLF input views as is

take_line does not strip CR, so the view is "data\r". It looks identical but eq is always false (one byte longer). Strip it with strings.remove_suffix line "\r" before comparing.

Counter-example. Forgetting attach · choosing loop continuation with guard

Spawning without attach is not a compile error but makes reads return none at once — zero lines from real input means check attach first. guard must leave, so using it to choose continuation is E-GUARD-FALLTHROUGH — choose with if.

Cautions. Do not hold views long — slices from take_line are not copies, so refilling the original buffer changes what they point to. Copy values you will keep. Ops calling actor handlers must declare state too. Why the attaching handler is attach, not open — it collided with files' open, and effect resolution picked the wrong one. Name things so they do not collide with modules used alongside.

`outbuf` — buffered output where forgetting to flush is a compile error

Source	`lib/out.low`
Layer	L2 — the outside world
Capabilities	`cap io` for `buf_flush` · `buf_write` · `buf_finish`

Output is **gathered and sent at once**. `write_out` flushes on every call, so calling it per piece costs a system call per piece — this writer accumulates bytes in a caller buffer and flushes only when full, reducing system calls from the number of bytes to the number of buffers. And it forces a final `finish`, preventing "ended without sending what was gathered" **at compile time**.

```
var p owned outbuf.pending be outbuf.buf_open 1 .
let w result (owned outbuf.pending) outbuf.io_error . be outbuf.buf_write out p buf "hi\n" .
guard is_ok w . else return 1 .
set p (ok_value w) .
let r result void outbuf.io_error . be outbuf.buf_finish out p buf .
```

What is owned is not the buffer but the bytes not yet sent. The buffer stays the caller's (the same discipline as `fmt`), and `owned` guards the unflushed state `pending` — exactly what must not be lost is what is owned. `finish` takes `owned pending` and returns a `result`, so silently dropping such a value is `E-OWN-INCOMPLETE` (chapter 19). This is inferred from signatures, not annotations. Elsewhere, forgetting to flush loses the output entirely, or a destructor flushes and discards the error — here it does not compile.

The writer does not hold `cap io`. A capability is a static token, not a value, and cannot be stored in a struct. So the caller supplies it at every point bytes actually leave — **the writer cannot flush behind your back** (chapter 16).

op	Shape	Failure
`write_all`	`proc (out cap io, d u64, b slice u8) → option u64` — writes one slice **to the end**, bypassing the buffer	`none` — cannot write any more
`io_error` · `pending`	enum `write_failed` · struct `pos u64` (bytes accumulated) · `fd u64` (1 = stdout, 2 = stderr)	—
`buf_open`	`fn (d u64) → pending`, effects none	none — but opening creates a debt
`buf_flush`	`proc (out cap io, p owned pending, buf mut slice u8) → result (owned pending) io_error`	`error write_failed`
`buf_write`	`proc (out cap io, p owned pending, buf mut slice u8, s slice u8) → result (owned pending) io_error` — flushes itself when full	`error write_failed`
`buf_finish`	`proc (out cap io, p owned pending, buf mut slice u8) → result void io_error` — flushes the tail and completes	`error write_failed`

Table 50.1 — Ops of `outbuf`

`buf_write` and `buf_flush` consume ownership and return the new state in `ok` — the caller takes it over with `set p (ok_value w)` every time. `set` is reinitialisation, not use, so ownership holds inside loops. **A partial write is failure** — if the count written differs from the request it is `write_failed`, the pending is consumed and bytes in the buffer at that moment are lost (retry was not built). A small buffer still behaves correctly; size only affects speed. Call `buf_flush` directly only when "this line must appear on screen now".

```
module emit .

use fmt .
use outbuf .

proc main input out cap io . input al cap allocator . output u8 . effects alloc io . do
  let g option mut slice u8 . . be alloc_bytes al capacity 16 .
  guard is_some g . else return 70 .
  let buf mut slice u8 . be some_value g .
  let ng option mut slice u8 . . be alloc_bytes al capacity 32 .
  guard is_some ng . else return 71 .
  let nb mut slice u8 . be some_value ng .
  var p owned outbuf.pending be outbuf.buf_open 1 .
  var i u64 be 1 .
  while le i 5 . do
    let a option u64 . be fmt.put_str nb 0 "line " .
    guard is_some a . else return 72 .
    let b option u64 . be fmt.put_u64 nb (some_value a) i .
    guard is_some b . else return 73 .
    let c option u64 . be fmt.put_nl nb (some_value b) .
    guard is_some c . else return 74 .
    let w result (owned outbuf.pending) outbuf.io_error .
      be outbuf.buf_write out p buf (subslice nb 0 (some_value c)) .
    guard is_ok w . else return 75 .
    set p (ok_value w) .
    set i (add i 1) .
  end
  let f result void outbuf.io_error . be outbuf.buf_finish out p buf .
  guard is_ok f . else return 76 .
  return 0 .
end
```

Pure formatting (`fmt`) assembles bytes and buffered output (`outbuf`) sends them. The assembly buffer `nb` and the writer's buffer `buf` are separate.

Counter-example. Omitting `finish` · reusing a moved `p`

Returning without completing an `owned pending` is `E-OWN-INCOMPLETE` — including paths leaving early with `return`. Passing the old `p` after `write` without `set p …` is a use of a moved value, a compile error. The compiler catches both.

Counter-example. Using the buffer for something else between `write` and `finish`

`pending.pos` counts bytes accumulated in that buffer. Sharing it with the assembly buffer compiles, but unsent bytes get overwritten and the wrong content goes out. It is the only silently wrong mistake in this module — dedicate one buffer to one writer.

Cautions. Flushing without a capability is a compile error (declaring `effects io` without receiving a capability is `E-EFFECT-NO-CAP`). The fd is an integer not checked for openness. If the

whole output is already in one buffer, no writer is needed — one `write_out` does it. Vectored writes (writev), retry after a failed flush and interruptible output were not built.

`files` — file and directory streams where forgetting to close is a compile error

Source	`lib/file.low`
Layer	L2 — the outside world
Capabilities	`cap file_system`

Opens, closes, reads and writes files and directories. Every op takes `cap file_system` first — the right to reach the file system is handed to `main` once, and only ops that receive it can touch files (chapter 16, chapter 28). The builtin leaves (`file_open`, `file_read`, `dir_open`, `file_type` and so on) only reach the kernel and know only integer handles. Holding a raw fd lets a forgotten close go unnoticed, so this module wraps it in **a value that cannot be forgotten** (`owned handle`).

```
let n result u64 files.file_error . be files.slurp fs "data.txt" buf .
```

op	Answer and why
`open` · `open_dir`	`result X file_error` — opening has no “end”. It opens or fails
`read` · `read_dir`	`result (option u64) file_error` — `ok (some n)` read · **`ok none` the end (not failure)** · `error e` failure
`write` · `seek_to` · `slurp`	`result u64 file_error` — no notion of “end”
`type_of`·`link_type_of`·`is_dir`·`is_file`·`is_symlink`	`option u64` · `bool` — the leaf gives “absent” and “stat failed” as one bit. No distinction that does not exist is promised by type

Table 50.1 — Answer shapes differ honestly per op

`file_error` variants are **only what can be distinguished** — `open_failed`, `read_failed`, `write_failed`, `seek_failed`, `close_failed`, plus `buffer_too_small`, which the library counts itself. The leaves give no errno, so the honest resolution stops at “which operation failed”.

op	What it does
`handle` · `dir_handle`	Unforgettable handles (one scalar `fd`) — files and directories are different types
`open fs path mode`	Opens. mode `0` read · `1` write (truncate and create) · `2` append
`read fs h buf` · `write fs h bytes`	Reads one piece (short reads are normal) · bytes written
`seek_to fs h off whence`	Moves the cursor. whence `0` absolute · `1` from current · `2` from end. `seek_to fs h 0 2` is the file size
`close fs h`	Consumes `owned handle`, returns `result void file_error`
`open_dir` · `read_dir fs h buf` · `close_dir`	Directory listing. `read_dir` puts the next name in the buffer (`.` and `..` included)
`type_of` · `link_type_of`	`some 0` file · `some 1` directory · `some 2` other · `none` if absent (`link_type_of` does not follow links)
`is_dir` · `is_file` · `is_symlink`	Conveniences — `false` if the path does not exist

`slurp fs path buf`	Whole file into the buffer. `error buffer_too_small` if it does not fit — never truncates

Table 50.2 — Ops of `files` — all `effects io`, first argument `fs cap file_system`

Forgetting to close does not compile. `close` and `close_dir` take `owned` and return `result`, so silently dropping a handle is `E-OWN-INCOMPLETE`. And closing really fails (NFS, full disks) — the `result` is not decoration. The handle parameter of `read` and `write` is non-owning, so reading many times and closing at the end is expressed as is.

Failure and EOF are different answers. `slurp` once just stopped its loop when a read failed, so callers could not tell complete reads from failures midway, and a line-counting program reported a read failure as the success "0 lines". A fault injector showed it concretely. Now failure answers `error` (the handle is still closed — releasing resources is separate from the answer).

```
proc main input fs cap file_system . input al cap allocator . input a cap args . output u8 .
effects alloc io . do
  let dpath option slice u8 . be arg a 0 .
  guard is_some dpath . else return 64 .
  let g option mut slice u8 . . be alloc_bytes al capacity 256 .
  guard is_some g . else return 70 .
  let buf mut slice u8 . be some_value g .
  let di result files.dir_handle files.file_error . be files.open_dir fs (some_value dpath) .
  guard is_ok di . else return 71 .
  var dh owned files.dir_handle be ok_value di .
  var total u64 be 0 .
  var going bool be true .
  while going . do
    let n result (option u64) files.file_error . be files.read_dir fs dh buf .
    if eq (is_ok n) false . do set going false . end
    if is_ok n . do
      let nvo option u64 . be ok_value n .
      if eq (is_some nvo) false . do set going false . end
      if is_some nvo . do set total (add total (some_value nvo)) . end
    end
  end
  let c result void files.file_error . be files.close_dir fs dh .
  guard is_ok c . else return 72 .
  return (narrow u8 total) .
end
```

Summoning failure yourself. `LOW_HOST_FAULT="open:err"` (every open fails), `"read:err@2"` (the second read fails), `"read:short@1=4"` (the first read cut to 4 bytes), `"close:err"`. VM and native use the same injector, so both back ends must answer the same.

> **Counter-example. Not closing on error paths**
>
> Once opened, early-return paths must close too. Missing it is `E-OWN-INCOMPLETE`, caught by the compiler.

> **Counter-example. Not filtering `read_dir` results**
>
> `.` and `..` come mixed in. Recursive traversal without filtering re-enters the same directory forever.

> **Counter-example. Treating `ok none` as failure · a short read as the end**
>
> `ok none` is the **fact** of the end. Conversely `ok (some n)` with fewer bytes than asked is not the end — keep reading (as `slurp` does).

Cautions. Paths are `slice u8` byte strings with no encoding interpretation (UTF-8 by convention). Reading the fd with `field h fd` and calling leaves directly loses what the owned discipline protects. If the buffer size is unknown, measure first with `seek_to fs h 0 2` or switch to a `read` loop. File system changes (making, removing, renaming) are stateless path → bool leaves and are not wrapped — pure forwarding would just be a synonym. Sockets belong to `net`. Input and output that can be interrupted while waiting (`wait`) does not exist yet — it can only be built once a reactor stands. Why `seek_to`, not `seek` — `seek` collided with another name.

`tty` — terminal input

Source	`lib/tty.low`
Layer	L2 — the capability half (builtins) + pure parsing (this module)
Capabilities	`cap tty` for the builtins `tty_raw` · `tty_read` · `tty_size`

For programs that **react to one key at a time**, like editors and menus. Normally a terminal hands lines over only after Enter; until then the kernel collects characters, echoes them and handles backspace (line buffering and echo). To move the cursor the moment an arrow key is pressed, that convenience gets in the way, so the terminal is switched to **raw mode**, receiving "immediately, without echo". And an arrow key is not one byte — `↑` is the three bytes `ESC [ A`, `Delete` the four bytes `ESC [ 3 ~`. `parse_key` reads such byte strings as keys.

```
let p option u64 . be tty.parse_key buf 0 .
guard is_some p . else return none .
let code u64 be tty.key_of (some_value p) .
let used u64 be tty.len_of (some_value p) .
```

Raw mode changes the user's terminal settings, and the change survives the program's death

If a program dies in raw mode, the user's shell is left without echo. If any code could call such a power, any library could break someone's shell. So only holders of `cap tty` may use it — calling `tty_raw` without the capability is `E-CAP-MISSING` at compile time. At the same time, putting key parsing behind the capability too would make it untestable without a real terminal. So it was split in half — **the pure half is much larger**. The capability half is just three: raw mode, reading and screen size; key parsing (here) and ANSI assembly, screen diffs and display width (`term`) are pure (chapter 37).

One keycode distinguishes characters from special keys. Ordinary bytes are their own value (1 … 255, control characters included — Ctrl-C is 3), and special keys are above 1000. So `is_char` is a single `lt keycode 1000`. **With no tuples**, the result packs into one value `keycode × 16 + length` (length 1 … 8). The length is needed to know where the next key starts in the buffer. Extract with `key_of` and `len_of` so code survives if the packing changes.

op	Layer	What it does
`key_up` · `key_down` · `key_right` · `key_left`	pure	1001 · 1002 · 1003 · 1004
`key_home` · `key_end` · `key_delete` · `key_pageup` · `key_pagedown` · `key_esc`	pure	1005 · 1006 · 1007 · 1008 · 1009 · 1010
`key_unknown`	pure	1011 — an unrecognised special key
`parse_key buf at`	pure (`effects none`)	byte string → `some (keycode × 16 + length)` or `none`
`key_of` · `len_of` · `is_char`	pure	keycode · length from a packed value · is it an ordinary character (pass a **keycode**)
`tty_raw t on` (builtin)	`cap tty`	enter (`true`) · leave (`false`) raw mode

`tty_read t dst` (builtin)	`cap tty`	bytes that have arrived into the buffer → `option u64` (`some 0` = nothing yet, not an error)
`tty_size t` (builtin)	`cap tty`	screen size → `option u64` (rows = `div v 4294967296`, columns = `mod v 4294967296`)

Table 50.1 — Ops of `tty` and the builtins

What `parse_key` recognises — one non-ESC byte (length 1), `ESC` with nothing after (the ESC key, length 1), `ESC O A–D · H · F` and `ESC [ A–D · H · F` (arrows, Home, End, length 3), `ESC [ 1 ~ · 3 ~ · 4 ~ · 5 ~ · 6 ~` (Home, Delete, End, PageUp, PageDown, length 4). **Incomplete sequences are not invented** — with only `ESC [` arrived it is `none`, not a guess at "probably an arrow". One exception — a lone `ESC` with nothing after is the ESC key (waiting forever is not an option). If `ESC` is followed by neither `[` nor `O`, it answers ESC key + length 1 (on terminals sending Alt combinations as ESC + character, the next call reads the character).

```
module keydemo .

use tty .

proc main input t cap tty . input al cap allocator . output u8 . effects alloc . do
  let g option mut slice u8 . . be alloc_bytes al capacity 32 .
  guard is_some g . else return 1 .
  let buf mut slice u8 . be some_value g .
  guard tty_raw t true . else return 1 .
  var going bool be true .
  var last u64 be 0 .
  while going . do
    let n option u64 . be tty_read t buf .
    guard is_some n . else do
      set going false .
      continue .
    end
    var off u64 be 0 .
    while lt off (some_value n) . do
      let p option u64 . be tty.parse_key (subslice buf 0 (some_value n)) off .
      guard is_some p . else do
        set off (some_value n) .
        continue .
      end
      let code u64 be tty.key_of (some_value p) .
      set last code .
      if eq code 113 . do set going false . end
      set off (add off (tty.len_of (some_value p))) .
    end
  end
  let r bool be tty_raw t false .
  return narrow u8 last .
end
```

Finish anything that can fail, like allocation, **before entering raw mode** — failing and returning after entering leaves the terminal raw. Terminal builtins take `cap tty` **as an argument** — the op holding the capability is not enough; it must be handed over where it is used. `parse_key`

is `effects none`, so all key parsing is verified by VM/native comparison without a terminal — the dividend of the split.

> **Counter-example. Not restoring raw mode**
>
> Leaving through a `guard … else return 2` after `tty_raw t true` skips restoration and leaves the shell broken. The language has no `defer`, so call `tty_raw t false` on every way out, or finish what can fail before entering raw mode. The example above is shaped so the loop always exits downwards.

> **Counter-example. Reading `none` as an error and quitting · advancing by 1 regardless of length**
>
> `none` from `parse_key` usually means "more bytes needed" — a read boundary can cut a sequence in the middle. Quitting there makes a program that dies on every arrow key. `set off (add off 1)` reads `ESC [ A` as three keys — always advance by `len_of`. Comparing the packed value directly with `1001` fails because the length is mixed in.

Cautions. Terminals differ in arrow sequences, so both `ESC [ A` and `ESC O A` are recognised. Sequences still not caught are `none` — never silently another key. Korean input comes from `parse_key` **one byte at a time**, so assemble code points with `utf8`. "Character" in `is_char` means "not a special key", not "printable". In raw mode Ctrl-C arrives as byte 3, not a signal. Screen size changes (SIGWINCH) are not reported — ask `tty_size` again each frame if needed. Mouse reports, bracketed paste and parsing terminal query replies were not built.

net — sockets (TCP loopback · in-process pairs)

Source	lib/net.low
Layer	L2 — the outside world
Capabilities	cap net

Opens, exchanges over and closes sockets. There are two ways to open — **TCP** (listening and connecting on 127.0.0.1) and a **pair** (two ends inside the same process, no address). Every op takes cap net. The ability to reach the world is a right handed over, not something fetched from somewhere (chapter 16).

Handles are resources. conn, listener and pair must be closed, and **closing can fail** (result void net_error). So forgetting to close does not compile (E-OWN-INCOMPLETE, chapter 19).

```
proc main input k cap net . output u8 . effects io . do
  let po option net.pair . be net.pair_of k .
  guard is_some po . else return 1 .
  var p owned net.pair be some_value po .
  let s option u64 . be net.send_all k (net.pair_a p) "hi" .
  guard is_some s . else return 2 .
  let c result void net.net_error . be net.shut_pair k p .
  guard is_ok c . else return 3 .
  return 0 .
end
```

op	Shape	Failure
serve	(cap net, port u64) → option listener	none — could not listen
dial	(cap net, port u64) → option conn	none — could not connect
take	(cap net, l listener) → option conn	none — could not accept
pair_of	(cap net) → option pair	none
send_all	(cap net, fd u64, buf slice u8) → option u64	answers **how much was sent**
recv_once	(cap net, fd u64, dst mut slice u8) → option u64	some 0 = peer closed · none = failure
shut · shut_listener · shut_pair	(cap net, owned X) → result void net_error	error close_failed. A pair closes both ends **separately**
conn_fd · pair_a · pair_b	→ u64	**pure** — no capability needed

Table 50.1 — Ops of net

The listening port is read with field l port (the assigned number if an ephemeral port was requested). Reading an fd is pure — a resource must not be forgotten, but looking at its number needs no right.

Two contracts to know. send_all sends to the end but does not lie — one send is not guaranteed to send everything, so it repeats, and if blocked midway it answers **how far it got**. On some n, check that n equals len buf. 0 from recv_once is not failure but the **fact** that the peer closed.

Why the close-three-at-once op was removed. Handles used to be integers, and shut3 closed three fds at once. Listening, connecting and accepted sockets are different resources, and

bundling three into one call leaves the compiler unable to say **what was forgotten**. Convenience that hides safety is not worth having. Now a pair holds each end as `owned conn`, so closing one end and forgetting the other does not compile.

Testing — summoning failure. `LOW_HOST_FAULT="connect:err"` (connection refused), `"accept:err"`, `"send:err@1"` (first send fails), `"recv:short=2"` (receive cut to 2 bytes). VM and native use the same injector, so both back ends must give the same answers (chapter 28).

Not built — remote targets (loopback only), timeouts, non-blocking I/O and polling, UDP, address parsing.

`clock` — time and deadlines

Source	`lib/clock.low`
Layer	L2 — the outside world
Capabilities	`cap clock`

Answers “what time is it”, “how long has passed” and “until when”.

Reading the clock is not `io`. In the specification `io` means exchanging data with the outside (files, connections, standard I/O), and reading the time exchanges no data. So clock ops declare no effect (`effects none`), and the fact that they break determinism is carried by a **capability** (`cap clock`). The one exception is `sleep_ms`, which has the `wait` effect. Declaring a larger effect would saddle callers with a cost that does not exist — pure ops could not use the clock. `cap random` is the pair split off for the same reason (chapter 16).

```
proc work input k cap clock . output u64 . effects none . do
  let due u64 be clock.deadline_in k 50 .
  var n u64 be 0 .
  while eq (clock.past k due) false . do set n (add n 1) . end
  return n .
end
```

	Monotonic	Wall clock (local)
ops	`now_ns` · `now_ms` · `since_ns` · `since_ms` · `sleep_ms` · `deadline_in` · `past` · `left_ms`	`local_packed` + `year_of` … `second_of`
Promises	**Never goes backwards** — so it measures durations	Human-readable date and time
Does not promise	Absolute time (the reference point is meaningless)	Monotonicity — time zones, daylight saving and corrections can move it backwards

Table 50.1 — Two clocks — similar names, different promises

Do not measure durations with the wall clock. That is why the table above exists.

op	Shape	Notes
`now_ns` · `now_ms`	`(cap clock) → u64`	monotonic
`since_ns` · `since_ms`	`(cap clock, start u64) → u64`	elapsed
`sleep_ms`	`(cap clock, ms u64) → u64`	time actually slept · `effects wait`
`deadline_in`	`(cap clock, ms u64) → u64`	deadline point
`past` · `left_ms`	`(cap clock, deadline u64) → bool · u64`	deadline passed · time left
`local_packed`	`(cap clock) → u64`	One value holding six wall-clock fields (year · month · day · hour · minute · second)
`year_of` ⋯ `second_of`	`(p u64) → u64`	**pure** — the unpacking pair

Table 50.2 — Ops of `clock`

Because the unpacking pair is pure, code that **receives** a time need not receive the clock capability.

What is checked — what can be claimed. Tests claim only monotonicity and ordering. Absolute time is not claimed because there is nothing to compare it with — tests do not invent distinctions that do not exist. **Not built** — time zones, calendar arithmetic (leap years, month ends), formatting, timers and alarms, the monotonic clock's reference point.

`trust` — finding a trusted root in the trust store

Source	`lib/trust.low`
Layer	L2 — host
Capabilities	`cap file_system`

The last question of chain checking is “was this intermediate signed by a certificate we **decided in advance to trust**”. Those certificates live in the **trust store** — on Linux, a bundle file of PEM certificates joined together. This module finds the one certificate in that bundle whose **subject name is this one**.

> **Here “trust” means “trust this file”**
>
> Who writes that file is not this module’s question; it belongs to whoever handed over the right to open it (`cap file_system`). To keep that visible the path is an **argument** (not built in).

It never holds the whole bundle

A system bundle is over 200 KB (measured: 224,449 bytes · 150 certificates). This language’s heap arena is 64 KiB. So it **streams** — it slides one window along, unwraps PEM blocks one at a time, compares only the name, and drops them. Only the one it finds is kept.

```
file:      [ cert 1 ][ cert 2 ][ cert 3 ][ cert 4 ] …
window 1:  [ cert 1 ][ cert 2 ][ front of cert 3 ]
                                └─ a block that is not complete is not unwrapped in this window
window 2:             [ front of cert 3 ][ rest ][ cert 4 ] …
                      └─ the leftover tail is pulled to the front, and the rest is filled by the
next read
```

Without pulling the tail forward, a certificate that straddles a window edge silently becomes “not there” — and that one may be exactly the root being looked for.

ops

op	what it does
`find_anchor`	finds the certificate whose subject name is `want` (at offset `wo`) in the bundle file `path`, puts its DER in `out` and returns the length. `effects io` — it reads a file

Table 50.1 — ops of `trust`

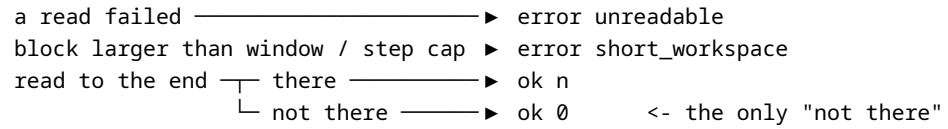
`find_anchor` has four answers.

answer	meaning
`ok n` (n > 0)	found — the first `n` bytes of `out` are that certificate
`ok 0`	read to the end, and it is not there — the only thing said **after measuring**
`error short_workspace`	the window (`win`, over 16 KiB) or the unwrap space was too small — it **could not read to the end** (a block larger than the window, or the step limit)

`error unreadable`	the bundle file could not be opened, or a read failed — check the path first

Table 50.2 — “not found” and “could not measure” are different

The last two rows once did not exist. A window too narrow to read everything gave 0, and a file that would not even open gave `ok 0` (fixed 2026-09-25). So the tool said “there is no trusted root” — which was not **absent** but **unmeasured**.



Linking the last step with the found root is `verify`’s `link_ok`. Modules it uses: `files` · `pem` · `x509` · `verify`.

Index

Only places where this book **defines or directly covers** a concept are gathered. Places where a word merely passes by are not listed.

Bernstein condition 509
Merkle DAG 533
RC11 515
abstract interpretation 477
actor 313
allocator 244
ambient authority 22
array 111
atomic operations 350
back-end cross-check 404
bit_cast 253
bitset 162
brand 440
build mode 179
build option 400
build tier 389
capability 196
cast 159
certificate recheck 491
channel 334
clause order 43
closer 36
completion 237
comptime 274
content addressing 533
contract 173
contract grades 178
denotation 526
effect 185
effect lattice 504
ensures 173
enum 122
errors clause 133
exclusivity rule 147
exhaustive checking 496
export 262
expr island 99
extern 375
fixed point 475, 499
fixed window 219
fn 67
generational handle 440
guard 87
heap 224
interrupt handler 388
interval analysis 485
invariant 476
join 474
lanes 352
leaf 412
local 77
mailbox 316
match 90
maturity 413
memory ordering 351
method 286
mmio 386
module 262
monomorphisation 276
narrowing 53
newtype 156
option 131
overflow 54
ownership 233
panic 91
parallel 347
partial order 474
pattern 138
pipe 299
pkg.low 398
prefix notation 36
proc 67
purity 68
range 158
reference 145
region 220
requires 173
result 133
same_slice 251
separation logic 520
sequential consistency 514
shadowing 42
short-circuit evaluation 101
slice 110
struct 121
task_group 333
trait 285
trusted base 538
try 135
using 247
widening 52