Lowent 매뉴얼

계약과 효과로 짜는 시스템 프로그래밍

초안(draft)

v0.2.0 · 최종 수정 2026-09-15

rubidus

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

이 책은 로우엔트 언어의 입문서이자 사용 설명서입니다.

대상 독자는 프로그래밍을 조금 해 본 사람부터,
C·Rust 같은 시스템 언어를 쓰는 사람까지입니다.

저작권과 연락처

지은이	rubidus
연락	rubidus@gmail.com
저장소	github.com/rubidus-api/lowent_lang
판	v0.2.0 — 초안(draft)
최종 수정	2026-09-15

본문 — 크리에이티브 커먼즈 저작자표시-비영리-동일조건변경허락 4.0 국제 라이선스(CC BY-NC-SA 4.0). 출처를 밝히면 자유롭게 공유하고 고칠 수 있으나, 영리 목적 이용은 허용되지 않으며, 고친 결과물에는 같은 라이선스를 적용해야 합니다.
https://creativecommons.org/licenses/by-nc-sa/4.0/
예제 코드 — MIT 라이선스. 자유롭게 가져다 쓰실 수 있습니다. 예제가 쓰는 로우엔트 컴파일러와 표준 라이브러리는 저장소의 MIT 라이선스를 따릅니다.

이 책의 모든 코드 시연은 실제로 `lowentc` 로 검사·실행해 얻은 출력을 그대로 인쇄한 것입니다. 실행하는 예제는 VM 과 네이티브 빌드가 같은 답을 내는지도 확인했습니다. 조판은 Typst로 했습니다.

로우엔트는 **실험 단계**의 언어입니다. 문법과 라이브러리가 바뀔 수 있으며, 이 책은 그 판에 맞춰 함께 고쳐집니다.

이 책은 계속 고쳐집니다. 지금 읽고 계신 것은 위 번호의 판이고, 그 뒤로도 오류 수정과 내용 보강이 이어집니다. 가장 새로운 판과 그동안의 변경 내역은 저자의 GitHub에 있습니다. 오래된 사본을 들고 계시다면 그쪽을 먼저 확인하시는 편이 좋습니다. 오류 신고와 수정 제안도 같은 자리에서 받습니다.

차례

머리말 16
이 책이 약속하는 것과 약속하지 않는 것 16
누구를 위한 책인가 16
판과 저작권 16
이 책을 읽는 법 17
길 ① — 이 언어를 처음부터 배운다 17
길 ② — Rust 나 C 에 익숙하고 차이만 알고 싶다 17
길 ③ — 운영체제 없는 기계에서 쓰려 한다 17
길 ④ — 이 언어의 주장이 어디까지 참인지 알고 싶다 17
본문의 장치 17

제1부 — 시작

1. Lowent 는 무엇을 하려는 언어인가 19
1.1 머리만 읽고 알 수 있는가 19
1.2 다섯 가지 생각 21
1.3 한 뜻에 한 표기 22
1.4 이 언어의 처지 22
1.5 이 책을 읽는 법 22
2. 첫 프로그램 — 짓고, 돌리고, 거절당하기 24
2.1 도구 짓기 25
2.2 hello, entropy 25
2.3 두 가지로 돌린다 26
2.4 계약이 멈추는 모습 27
2.5 거절당하기 27
2.6 모양을 고쳐 주는 `--fmt` 28
2.7 인자와 권한 29
2.8 흔한 실수 29
2.9 이 장의 문법 한눈에 32
3. 겉모습 — 마침표, 블록, 절의 차례 33
3.1 이름이 먼저, 인자가 뒤 34
3.2 떨어진 마침표가 닫는다 34
3.3 블록은 언제나 `do … end` 36
3.4 주석과 리터럴 36
3.5 이름 39
3.6 낱말은 예산이다 40
3.7 `expr` 섬의 경계 40
3.8 op 머리의 절 차례 40
3.9 흔한 실수 41
3.10 이 장의 문법 한눈에 45

제2부 — 값과 흐름

4. 수 — 폭이 정해진 정수와 부동소수 47
4.1 정수 타입은 이름에 폭을 적는다 48
4.2 넓히기는 자동, 좁히기는 적는다 48
4.3 넘치면 멈춘다 49
4.4 나눗셈과 나머지 51
4.5 참거짓은 수가 아니다 52

4.6 비트 연산 ... 52
4.7 부동소수 ... 53
4.8 흔한 실수 ... 55
4.9 이 장의 문법 한눈에 ... 60

5. op — `fn` 과 `proc` ... 62
5.1 두 갈래 ... 62
5.2 순수함은 관측으로 정한다 ... 63
5.3 `fn` 에는 `effects` 를 적지 않는다 ... 64
5.4 효과는 부르는 쪽으로 번진다 ... 65
5.5 매개변수와 반환 ... 66
5.6 `neg` 만 단항이다 ... 66
5.7 수식어 ... 67
5.8 흔한 실수 ... 67
5.9 이 장의 문법 한눈에 ... 70

6. 지역 — `let` 과 `var` ... 72
6.1 바뀌는지는 낱말이 말한다 ... 72
6.2 타입은 적거나, 값에게 맡긴다 ... 74
6.3 값 없는 이름은 없다 ... 74
6.4 지역이 사는 범위 ... 75
6.5 `if` 는 값을 내지 않는다 ... 76
6.6 흔한 실수 ... 77
6.7 이 장의 문법 한눈에 ... 79

7. 흐름 — 갈래, 되풀이, 빠져나가기 ... 80
7.1 조건과 되풀이 ... 80
7.2 `guard` — 조건을 사실로 바꾼다 ... 82
7.3 모든 길이 값을 돌려준다 ... 83
7.4 `match` — 경우를 빠짐없이 ... 84
7.5 `panic` 은 효과다 ... 86
7.6 흔한 실수 ... 86
7.7 이 장의 문법 한눈에 ... 91

8. 식 — 전위 표기와 `expr` 섬 ... 92
8.1 전위 표기에는 외울 것이 없다 ... 92
8.2 `expr` 섬의 우선순위표 ... 94
8.3 섬 안에서 하지 못하는 것 ... 94
8.4 단락 평가로 조건을 지킨다 ... 95
8.5 번역할 때 계산되는 식 ... 96
8.6 흔한 실수 ... 97
8.7 이 장의 문법 한눈에 ... 100

제3부 — 데이터

9. 줄 — 배열과 슬라이스 ... 102
9.1 시작과 길이를 함께 갖는다 ... 102
9.2 쓰려면 `mut slice` ... 104
9.3 창을 좁힌다 ... 105
9.4 계약이 경계 검사를 지운다 ... 106
9.5 흔한 실수 ... 108
9.6 이 장의 문법 한눈에 ... 112

10. 묶음 — `struct` 와 `enum` ... 113

10.1 `struct` — 이름 붙은 칸의 모음 113
10.2 `enum` — 여럿 중 하나 115
10.3 갈래는 마침표로 닫는다 116
10.4 자기 자신을 품을 수 없다 117
10.5 흔한 실수 118
10.6 이 장의 문법 한눈에 122

11. 답을 담는 타입 — `option` 과 `result` 123
11.1 `option` — 값이 있거나 없거나 124
11.2 `result` 와 `errors` 절 125
11.3 `try` — 실패를 위로 넘긴다 127
11.4 두 채널 사이를 건넌다 127
11.5 실패를 말하는 세 길 128
11.6 패턴을 묶고 겹친다 129
11.7 흔한 실수 131
11.8 이 장의 문법 한눈에 135

12. 빌리기 — `ref` 와 `mut_ref` 136
12.1 두 가지 빌림 136
12.2 쓰기 권한은 한 방향으로만 좁아진다 138
12.3 읽기 여럿 또는 쓰기 하나 138
12.4 빌림은 빌려준 값보다 오래 살 수 없다 140
12.5 `mut ref slice` 는 없다 140
12.6 흔한 실수 141
12.7 이 장의 문법 한눈에 144

13. 이름 붙인 타입 — `type·newtype·range·cast` 146
13.1 `type` 은 별칭, `newtype` 은 새 타입 147
13.2 `range` — 매개변수의 생김새가 된 계약 149
13.3 `cast` — 값이 변할 수 있는 자리 150
13.4 `bits` — 1 부터 64 비트까지 151
13.5 바이트 배치를 못 박는다 151
13.6 배치의 두 방향과 실패하는 뷰 152
13.7 `bitset` — 작은 수의 집합 153
13.8 흩어진 조각을 한 눈으로 — `segments` 155
13.9 타입 낱말의 세 갈래 155
13.10 흔한 실수 157
13.11 이 장의 문법 한눈에 159

제4부 — 계약과 효과

14. 계약 — 적고, 검사받고, 검사를 지운다 162
14.1 계약은 검사되는 약속이다 163
14.2 누구의 잘못인가 163
14.3 계약이 검사를 지운다 165
14.4 모든 원소에 대한 조건 165
14.5 계약에 이름을 준다 166
14.6 지금 판정할 수 있으면 지금 판정한다 166
14.7 계약의 등급 168
14.8 빌드 모드가 남은 검사를 정한다 169
14.9 흔한 실수 170
14.10 이 장의 문법 한눈에 172

15. 효과 — op 이 세상에 남기는 자국 174

15.1 효과는 닫힌 낱말이다 ... 174
15.2 효과는 부르는 쪽으로 번진다 ... 175
15.3 적어 놓고 안 하면 알린다 ... 177
15.4 효과 줄은 집합이다 ... 177
15.5 `via` — 타입 인자가 효과를 정한다 ... 178
15.6 순수함이 허락하는 것 ... 179
15.7 흔한 실수 ... 179
15.8 이 장의 문법 한눈에 ... 183

16. 권한 — 건네받는 힘 ... 185
16.1 권한의 종류 ... 185
16.2 권한은 사슬을 타고 내려간다 ... 186
16.3 효과와 권한은 짝이다 ... 187
16.4 있음이 아니라 종류다 ... 189
16.5 시작점이 받는 것 ... 190
16.6 흔한 실수 ... 191
16.7 이 장의 문법 한눈에 ... 194

17. 실패를 설계하기 ... 195
17.1 경계와 안쪽 ... 196
17.2 오류 열거형을 나누는 기준 ... 197
17.3 층을 올라가며 ... 198
17.4 정보를 버려도 되는 자리 ... 199
17.5 `panic` 을 써도 되는 자리 ... 199
17.6 흔한 실수 ... 200
17.7 이 장의 문법 한눈에 ... 204

제5부 — 메모리

18. 영역 — 값이 사는 곳과 한꺼번에 걷히는 메모리 ... 206
18.1 값이 사는 세 곳 ... 207
18.2 영역을 열고 자리를 얻는다 ... 207
18.3 영역을 건네받는다 ... 209
18.4 영역 밖으로 들고 나갈 수 없다 ... 210
18.5 뿌리마다 커서는 하나다 ... 211
18.6 자라는 뿌리 ... 212
18.7 영역 위의 스택 ... 213
18.8 흔한 실수 ... 214
18.9 이 장의 문법 한눈에 ... 218

19. 소유 — 없앨 책임은 하나에게 ... 220
19.1 `owned` 와 `drop` ... 221
19.2 갈래가 만나는 자리 ... 223
19.3 해제와 완결 ... 223
19.4 운영체제가 치워 주기를 기대하지 않는다 ... 225
19.5 메모리 안전은 어느 강도로 보장되나 ... 226
19.6 흔한 실수 ... 226
19.7 이 장의 문법 한눈에 ... 229

20. 할당기와 고정 메모리 ... 231
20.1 할당기를 이루는 셋 ... 232
20.2 빌린 바이트를 잘라 준다 ... 232
20.3 할당기를 갈아 끼운다 ... 233
20.4 뿌리에서 곧장 깎는 기본 할당기 ... 236

20.5 조각을 늘리고 돌려준다 — `grow` 와 `release` ... 237
20.6 표준 라이브러리에서 셋은 어디에 있나 ... 239
20.7 고정 창의 크기는 누가 정하나 ... 239
20.8 비트는 그대로, 읽는 법만 ... 240
20.9 흔한 실수 ... 241
20.10 이 장의 문법 한눈에 ... 245

제6부 — 추상

21. 모듈 — 감춘 것이 기본이다 ... 248
21.1 파일 하나, 모듈 하나 ... 249
21.2 들여오기 ... 250
21.3 무엇을 내보낼 수 있고, 어떻게 줄여 부르나 ... 251
21.4 검색 경로가 없다 ... 252
21.5 이름이 부딪히면 ... 253
21.6 최상위 선언의 차례는 무관하다 ... 253
21.7 흔한 실수 ... 254
21.8 이 장의 문법 한눈에 ... 259
22. 제네릭 — 번역 시점에 정해지는 매개변수 ... 261
22.1 값과 타입을 번역 시점에 받는다 ... 261
22.2 조합마다 실물이 만들어진다 ... 263
22.3 타입에 조건을 건다 ... 263
22.4 값을 넘기지 않고 타입이 들고 온다 ... 265
22.5 흔한 실수 ... 266
22.6 이 장의 문법 한눈에 ... 270
23. 트레이트 — 여러 타입이 지키는 한 가지 약속 ... 271
23.1 무엇에 쓰나 ... 272
23.2 타입에 붙은 op 과 `method` ... 273
23.3 op 을 여럿 가진 트레이트 ... 273
23.4 서명에는 `fn`·`proc` 을 적지 않는다 ... 275
23.5 갖추지 못했을 때 ... 277
23.6 `via self` — 할당 효과만 더 적을 수 있다 ... 278
23.7 흔한 실수 ... 279
23.8 이 장의 문법 한눈에 ... 283
24. `pipe` — 하려는 일만 한 줄씩 ... 284
24.1 같은 일, 두 가지 모양 ... 285
24.2 스테이지와 종결자 ... 286
24.3 한 번에 훑는 것은 정의다 ... 290
24.4 필요한 만큼만 읽는다 ... 290
24.5 이름이 같은 내장 `map`·`filter` ... 291
24.6 흔한 실수 ... 292
24.7 이 장의 문법 한눈에 ... 295

제7부 — 동시성

25. 액터 — 상태를 가진 채 메시지로 사는 것 ... 298
25.1 선언하고, 만들고, 말을 건다 ... 299
25.2 메시지에 값을 싣는다 ... 300
25.3 우편함 — 넣기와 비우기 ... 302
25.4 터지게 두고 다시 세운다 ... 303
25.5 액터를 쓸 수 있는 자리 ... 304

25.6 권한을 든 액터와 상태에 둘 수 있는 것 305
25.7 액터로 설계하기 — 계좌 둘 사이의 이체 307
25.8 흔한 실수 308
25.9 이 장의 문법 한눈에 315

26. 태스크와 채널 — 묶인 흐름끼리 주고받기 317
26.1 흐름은 블록에 묶인다 318
26.2 채널 — 차례 있는 그릇 319
26.3 적힌 것만 봐도 아는 교착 321
26.4 흐름과 메모리 323
26.5 아직 없는 것 325
26.6 흔한 실수 326
26.7 이 장의 문법 한눈에 329

27. 병렬 되풀이와 원자 연산 331
27.1 나눌 수 있다고 밝힌다 332
27.2 처리기가 확인하는 세 조건 333
27.3 모으는 연산은 결합적이어야 한다 334
27.4 원자 연산 335
27.5 기억 차례 336
27.6 레인 — 값 여럿을 한 번에 셈한다 337
27.7 자리를 어떻게 쓰는지 적는다 — `access` 341
27.8 흔한 실수 341
27.9 이 장의 문법 한눈에 345

제8부 — 바깥 세계

28. 입출력과 파일 348
28.1 표준출력 348
28.2 파일을 통째로 읽는다 349
28.3 열었으면 닫는다 350
28.4 끝과 실패는 다른 답이다 351
28.5 실패를 일부러 일으킨다 352
28.6 흔한 실수 353
28.7 이 장의 문법 한눈에 356

29. C 와 만나는 자리 358
29.1 C 를 부른다 — 표시·권리·효과 줄 358
29.2 경계를 건너는 타입 360
29.3 C 가 Lowent 를 부른다 361
29.4 되부름과 소유 362
29.5 흔한 실수 364
29.6 이 장의 문법 한눈에 367

30. 하드웨어 — 레지스터, 인터럽트, 기계 명령 369
30.1 레지스터 지도 370
30.2 닿는 법은 번역에서 강제된다 371
30.3 인터럽트 처리기 372
30.4 기계가 감당하는 것 — `build tier` 373
30.5 기계 명령 — `asm` 374
30.6 흡수 경계 — `unsafe` 가 멈추는 자리 375
30.7 기계가 가진 것을 쓸까 — `--hw` 376
30.8 흔한 실수 377
30.9 이 장의 문법 한눈에 380

31. 짓고 시험하기 — 패키지, 구성, 시험, 대조 382
31.1 매니페스트와 프로젝트 383
31.2 의존은 해시로 고정한다 383
31.3 빌드마다 다른 프로그램 — `build option` 과 `config` 384
31.4 시험 386
31.5 컴파일러를 검증하는 법 388
31.6 느린 자리를 묻는다 389
31.7 흔한 실수 389
31.8 이 장의 문법 한눈에 392

제9부 — 표준 라이브러리

32. 표준 라이브러리의 지도 394
32.1 세 층 — 언어, 잎, 라이브러리 395
32.2 무엇이 들어오는가 395
32.3 층으로 본 지도 396
32.4 파일 이름과 모듈 이름 396
32.5 모든 모듈이 따르는 규약 397
32.6 잎 op 의 목록 — 모듈 아래에 무엇이 있나 398
32.7 흔한 실수 400
32.8 이 장의 문법 한눈에 401

33. 글자와 부호화 — `strings·fmt·utf8·codec·hash` 403
33.1 자르고 찾고 조립한다 403
33.2 UTF-8 — 치환 문자를 만들지 않는다 405
33.3 16진과 base64 406
33.4 해시 — 세 가지 물음에 세 가지 답 406
33.5 정규식 — 역추적하지 않는다 407
33.6 흔한 실수 407
33.7 이 장의 문법 한눈에 410

34. 그릇과 정렬 — `sortlib·sortgen·hashmap·vecgen·spsc` 411
34.1 `u64` 를 정렬하고 찾는다 412
34.2 타입이 기준을 들고 온다 412
34.3 호출자의 슬라이스 위의 해시맵 413
34.4 자라는 제네릭 벡터 414
34.5 흐름 사이로 값을 넘기는 링 버퍼 415
34.6 그 밖의 그릇 415
34.7 흔한 실수 416
34.8 이 장의 문법 한눈에 419

35. 저장소와 핸들 — `pool·shard·budget·wire` 420
35.1 세대 핸들 421
35.2 브랜드 — 풀을 섞지 못하게 422
35.3 겹치지 않는 조각 — `shard` 422
35.4 비트 예산 — `budget` 423
35.5 비트 칸 — `wire` 424
35.6 그 밖의 저장소 모듈 424
35.7 흔한 실수 425
35.8 이 장의 문법 한눈에 427

36. 입출력·네트워크·시간·난수·암호 429
36.1 버퍼링 출력 — 비우기를 잊으면 번역이 거절한다 430
36.2 네트워크 — 핸들은 자원이다 431

36.3 난수 — 재현되는 열과 운영체제 엔트로피 432
36.4 시간 — 단조 시계와 벽시계 432
36.5 HTTP 요청 파서 — 알맹이는 거절이다 433
36.6 암호 — 쌓인 차례와 아직 없는 꼭대기 434
36.7 흔한 실수 434
36.8 이 장의 문법 한눈에 437

37. 터미널 — `term` 과 `tty` 438
37.1 `term` 은 화면에 쓰지 않는다 439
37.2 바뀐 칸만 다시 그린다 440
37.3 칸의 수는 바이트의 수가 아니다 440
37.4 키를 읽는 것은 계산이다 441
37.5 raw 모드는 권한이다 442
37.6 무엇을 아직 안 지었나 443
37.7 흔한 실수 443
37.8 이 장의 문법 한눈에 445

제10부 — 근거: 무엇이 증명되었나

38. 왜 증명하는가 448
38.1 시험은 있음을, 증명은 없음을 448
38.2 세 층을 겹친다 450
38.3 무엇이 증명되어 있나 450
38.4 증명하는 세 가지 방법 451
38.5 이 부를 읽는 약속 452

39. 수학 도구상자 453
39.1 여섯 가지 도구 454
39.2 집합과 관계 454
39.3 부분 순서 — 비교할 수 없는 쌍 454
39.4 격자와 join — 합칠 때 무엇을 고르나 454
39.5 단조 함수와 고정점 — 분석은 왜 멈추는가 455
39.6 귀납법과 불변식 — 도미노 456
39.7 추상해석 — 값 대신 범위로 456
39.8 기호 표 457
39.9 도구들이 맞물리는 방식 457

40. 수의 증명 — 넓히기, 좁히기, 나눗셈 459
40.1 넓히기는 부분 순서다 459
40.2 나눗셈과 나머지 461
40.3 `range` 가 주는 정리 462
40.4 멈춤을 정직하게 모델에 넣는다 463
40.5 증명하지 않은 것 463

41. 경계의 증명 — 구간, 관계, 행우선 주소 465
41.1 구간으로 계산한다 466
41.2 관계 — 구간이 못 하는 것 467
41.3 네 단계 467
41.4 행우선 증명 469
41.5 증명을 신뢰하되, 신뢰를 검사로 받친다 471
41.6 증명하지 않은 것 471

42. 소유와 차용의 증명 473
42.1 두 기계와 세 사건 474

42.2 중심 정리 ... 474
42.3 거절되는 모양, 통과하는 모양 ... 475
42.4 모델을 넓히다가 구현의 결함을 찾았다 ... 475
42.5 모델과 구현 사이 ... 476
42.6 증명하지 않은 것 ... 477

43. 되풀이와 고정점 — “몇 번을 돌든” 을 증명하기 ... 478
43.1 두 번째 회차에서 깨진다 ... 478
43.2 고정점이면 몇 번이든 ... 480
43.3 위험한 모양은 하나뿐이다 ... 480
43.4 신선한 토큰 — 이름 문제를 없애는 한 수 ... 480
43.5 중첩 반복 — 함수를 관계로 ... 481
43.6 증명하지 않은 것 ... 482

44. 효과의 증명 — “무엇을 할 수 있나” 를 타입에 적기 ... 483
44.1 효과는 원자의 집합이다 ... 484
44.2 처리기가 강제하는 네 규칙 ... 484
44.3 권한 — 효과의 짝 ... 486
44.4 효과 건전성 — 순환이 아니다 ... 486
44.5 효과에 기댄 최적화는 적법하다 ... 487
44.6 증명하지 않은 것 ... 487

45. 경합과 병렬의 증명 — 규율이 메모리 모델을 대신한다 ... 489
45.1 Bernstein 조건 — 1966 년의 답 ... 490
45.2 독립이면 경합이 없다, 그리고 그 조건이 필요하다 ... 490
45.3 액터 사이에는 반드시 메시지가 있다 ... 491
45.4 가둠 — 무거운 도구가 필요한 곳 ... 491
45.5 경합이 없는 것과 결정적인 것은 다르다 ... 491
45.6 증명하지 않은 것 ... 492

46. 약한 메모리의 증명 — 기본값이면 순서대로 생각해도 된다 ... 494
46.1 순차 일관성과 그 반례 ... 495
46.2 실행은 그래프다 ... 495
46.3 기본값이면 나올 수 없고, `relaxed` 면 나올 수 있다 ... 495
46.4 막는 것은 정확히 그 공리다 ... 496
46.5 메시지 전달 — 약한 것은 이 모양을 위해 있다 ... 496
46.6 모든 실행에 대해 ... 497
46.7 약하게 적으면 거동이 늘어난다 ... 497
46.8 문헌의 시험 일곱 ... 498
46.9 증명하지 않은 것 ... 498

47. 락의 증명 — 무거운 도구가 실제로 필요한 자리 ... 500
47.1 분리논리의 어휘 ... 501
47.2 토큰은 둘일 수 없다 ... 501
47.3 도구를 필요한 만큼만 ... 502
47.4 rwlock — 같은 문장의 두 증명 ... 502
47.5 교착 자유와 굶주림 없음 ... 502
47.6 약한 메모리에서 빌린 증명 — `spsc` ... 503
47.7 증명하지 않은 것 ... 504

48. 문법의 증명 — 어느 닫개로 닫아도 같은 나무 ... 506
48.1 대수와 뜻함수 ... 507
48.2 한 문장은 한 문장짜리 블록이다 ... 507
48.3 닫개는 같은 일을 한다 ... 508

48.4 흔한 실수 510
48.5 증명하지 않은 것 511
49. 해시의 증명 — 이름이 아니라 내용으로 부르기 512
49.1 해시 함수와 Merkle DAG 512
49.2 이름이 아니라 내용으로 부른다 513
49.3 무엇이 해시를 바꾸고 무엇이 안 바꾸나 513
49.4 증명할 수 있는 것과 없는 것 515
49.5 해시가 막는 것 516
49.6 증명하지 않은 것 516
50. 증명하지 않은 것 518
50.1 신뢰 기반 — 증명이 기대는 것 518
50.2 가장 큰 간극 — 모델과 구현 사이 519
50.3 남은 것 — 주제별로 520
50.4 “전면 정적 안전” 으로 팔지 않는다 521
50.5 그래서 무엇을 믿어도 되나 521

부록과 찾아보기

부록 A — 낱말과 내장 연산 524
낱말 마흔셋 524
없앤 낱말 524
자주 쓰는 내장 연산 524
효과와 권한 526
부록 B — 진단 찾아보기 527
번역할 때 527
실행 중에 멈출 때 531
그 밖의 진단 532
부록 C — 흔한 실수와 고치는 법 534
겉모습 534
값과 흐름 534
효과·권한·메모리 535
추상과 동시성 535
부록 D — 문법 요약 536
op 머리의 절 차례 536
선언 536
문장 536
패턴 537
식 537
머리와 닫개 537
부록 E — 표준 라이브러리 모듈 538
`strings` — 문자열 뷰 연산 541
설계와 경계 541
op 한눈에 541
op 상세 542
쓰는 법 542
반례 543
주의 544
`strbuf` — 소유 문자열 버퍼와 널 종단 `cstr` 545

설계와 경계 545
op 한눈에 546
op 상세 546
쓰는 법 546
반례 547
주의 548
fmt — 호출자의 버퍼에 조립하는 포매팅 549
설계와 경계 549
op 한눈에 549
op 상세 550
쓰는 법 550
반례 551
주의 551
utf8 — UTF-8 코드포인트 순회와 검증 553
설계와 경계 553
op 한눈에 554
op 상세 554
쓰는 법 554
반례 555
주의 556
utf16 — UTF-16 서로게이트 산술 557
설계와 경계 557
op 한눈에 558
op 상세 558
쓰는 법 559
반례 560
주의 560
unicode — 유니코드 속성 표 561
설계와 경계 561
op 한눈에 562
op 상세 562
쓰는 법 563
반례 564
주의 564
codec — 16 진과 base64 565
설계와 경계 565
op 한눈에 566
op 상세 566
쓰는 법 566
반례 567
주의 568
regex — 역추적하지 않는 Pike VM 정규식 569
설계와 경계 569
자료구조 570
op 한눈에 570
op 상세 571
쓰는 법 571
반례 573

주의 573
term — 터미널 렌더러(순수 절반) 574
sortlib — 제자리 quicksort 577
sortgen — 제네릭 정렬(비교는 타입이 들고 온다) 579
searchlib — 정렬된 슬라이스의 이진 탐색 581
hashmap — u64 → u64 해시 맵 583
strmap — 문자열 키 해시 맵 585
vecs — 호출자가 버퍼를 드는 성장 바이트 벡터 587
spsc — 락 없는 SPSC 링 버퍼 589
hash — 해시(해시맵 자리 · 손상 검출 · SHA-256) 590
math — 부동소수 수학 591
random — 난수(재현되는 열 · OS 엔트로피) 592
hmac — HMAC-SHA256 과 HKDF 593
chacha — ChaCha20 스트림 암호 594
poly — Poly1305 한 번 쓰는 인증자 595
aead — ChaCha20-Poly1305 봉인과 개봉 596
x25519 — 곡선 위의 키 합의 597
aes — AES-128 블록 암호 598
gcm — AES-128-GCM 인증 암호 599
crypto_hw — 기계가 도와주는 암호 셈 600
bigint — 큰수 모듈러 산술 601
rsa — RSASSA-PSS 검증 602
p256 — NIST P-256 곡선과 ECDSA 검증 603
p384 — NIST P-384 곡선과 ECDSA 검증 604
ecdsa — ECDSA P-256 서명 생성 606
ed25519 — Ed25519 서명 검증 607
der — DER 최소 파서 608
pem — PEM 봉투 벗기기 609
x509 — X.509 인증서 읽기 610
verify — 인증서 서명과 체인의 한 마디 확인 612
체인은 마디의 줄이다 612
op 612
설계 613
tls13 — TLS 1.3 의 계산 부품 614
tlssrv — TLS 1.3 서버 핸드셰이크 616
tlscli — TLS 1.3 클라이언트 핸드셰이크 618
핸드셰이크의 차례 618
op 618

설계 619
http — HTTP/1.1 요청 파서 620
soa — SoA 배치 시범: 필드마다 배열 하나 622
allocs — 얼로케이터 trait 과 범프 · 기본 얼로케이터 624
pool — 세대 핸들 블록 풀 627
shard — 저장소를 쪼개는 접근 단위 630
budget — 핸들 비트 예산과 세대 한 바퀴 632
wire — 한 낱말을 칸으로 나눠 쓰기 634
flags — 이름 붙은 켬 · 끔을 한 낱말에 636
segarena — 고정 크기 세그먼트 아레나 638
pagecache — 페이지 id 와 고정 커서 640
growvec — 자가성장 바이트 벡터 642
vecgen — 제네릭 자가성장 벡터 vec t a 644
mapgen — 제네릭 해시맵 table k v 646
nodelist — 고정 intrusive 목록 648
segview — 조각 뷰의 커서 · 총길이 · 펴기 649
lifemode · lifeatom — 값이 언제 끝나는가 650
io — 슬라이스 위의 스트림 읽기 651
outbuf — 버퍼링 출력, flush 망각은 컴파일 오류 653
files — 파일 · 디렉터리 스트림, close 망각은 컴파일 오류 656
tty — 터미널 입력 659
net — 소켓(TCP loopback · 프로세스 안 한 쌍) 662
clock — 시각과 마감 664
trust — 신뢰 저장소에서 믿는 뿌리 찾기 665
묶음을 통째로 들지 않는다 665
op 665
찾아보기 667

머리말

이 책은 Lowent 라는 실험 중인 시스템 프로그래밍 언어의 안내서다. 저장소에 있던 매뉴얼을 한 권의 책으로 다시 엮었다. 흩어져 있던 매뉴얼 장들과 명세의 설명, 표준 라이브러리 문서, 증명 해설을 한 줄기의 순서로 늘어놓고, 모든 예제를 실제로 돌려 확인했다.

Lowent 가 풀려는 문제는 한 문장이다. **op 의 머리만 읽고 그 op 이 할 수 있는 일과 못 하는 일을 알게 한다.** 순수한지, 무엇을 받고 무엇을 돌려주는지, 바깥에 무엇을 하는지, 누가 그것을 허락했는지를 머리에 적게 하고, 컴파일러가 그것을 검사한다. 그 목표가 이 언어의 낯선 겉모습 — 마침표로 닫는 절, 인자로 건네받는 권한, 한 가지 차례로만 적는 머리 — 을 거의 다 설명한다. 그래서 이 책은 문법보다 그 목표를 먼저 세우고, 규칙마다 그 목표에서 어떻게 나왔는지를 설명하려 했다.

이 책이 약속하는 것과 약속하지 않는 것

이 책의 예제는 모두 `docs/manual/examples/` 에 있는 실제 파일이다. 파일마다 첫머리에 무엇을 해야 하는지가 적혀 있고 — VM 과 네이티브로 돌려 같은 답을 내야 하는지, 어떤 진단으로 거절되어야 하는지 — 검증 스크립트가 그것을 확인한 뒤에 남긴 출력을 지면에 그대로 싣는다. 사람이 옮겨 적은 실행 결과는 없다.

그 규율 덕분에 이 책을 쓰는 동안 컴파일러와 명세 사이의 어긋남이 여럿 드러났다. 고칠 수 있는 것은 예제를 바꾸어 피했고, 독자가 걸릴 만한 것은 본문에 결함으로 적었다. 그러니 이 책은 **지금 저장소의 컴파일러가 실제로 받아들이는 언어**를 설명한다. 명세가 적은 이상과 다른 자리가 있으면 그 사실을 적는다.

반대로 약속하지 않는 것도 있다. 이 언어는 버전 1 에 이르지 않았고, 문법과 표준 라이브러리는 아직 바뀐다. 이 책은 판 번호가 따로 있고, 컴파일러가 바뀌면 책도 따라 바뀐다. 그리고 이 책은 연습문제를 두지 않는다. 설명하고, 질문을 던지고, 답한다. 실제로 코드를 만들어 내는 힘은 직접 써 보는 일로만 자란다. 다행히 이 언어의 컴파일러는 틀린 코드를 코드가 붙은 진단으로 거절하므로, 혼자 써 보는 일이 다른 언어보다 덜 외롭다.

누구를 위한 책인가

프로그래밍을 한 가지 이상 해 본 독자를 가정한다. 변수와 반복과 함수가 무엇인지는 설명하지 않고, Lowent 에서 그것이 **어떻게 다른지**를 설명한다. C 를 알면 메모리와 바깥 세계를 다루는 부가 쉽고, Rust 를 알면 차용 규칙이 익숙할 것이다. 둘 다 몰라도 읽을 수 있게 썼다. 제10부의 증명 해설은 고등학교 수학이면 따라온다.

판과 저작권

이 책의 글은 CC BY-NC-SA 4.0 으로, 예제 코드는 MIT 로 공개한다. 책은 한국어판과 영어판이 함께 나오고, 두 판의 장 차례와 예제는 같다. 예제의 코드와 출력은 두 판이 한 벌을 나누어 쓴다.

이 책을 읽는 법

이 책은 **앞에서 뒤로 읽는 것**을 기준으로 쓰였다. 뒤의 장은 앞의 장이 세운 어휘와 규칙 위에 선다. 각 장의 첫머리에 "먼저 알아야 할 것" 이 장 번호와 함께 적혀 있으므로, 건너뛰어 읽다가 막히면 그 자리로 돌아가면 된다.

다만 읽는 사람의 사정은 저마다 다르다. 아래의 갈래는 자기 자리에 가까운 것을 골라 쓰는 **단축 경로**다.

★ 어느 경로를 고르든 **제1부와 제4부**는 덜어 내지 않는다. 제1부는 이 언어의 목표와 겉모습을, 제4부는 그 목표를 받치는 계약·효과· 권한을 세운다. 나머지 모든 부가 이 둘 위에 서 있다.

길 ① — 이 언어를 처음부터 배운다

책의 차례 그대로다. 제1부부터 제8부까지 차례로 읽고, 제9부는 필요한 장만, 제10부는 마지막 장(50장)만이라도 읽는다.

길 ② — Rust 나 C 에 익숙하고 차이만 알고 싶다

1–3장 → 4·11장 → 제4부 전체(14–17장) → 제5부(18–20장) → 22·23장 → 27장 → 29장. 제2·3부의 나머지는 각 장의 "복습 정리" 만 훑고, 처음 보는 규칙이 있으면 그 장을 읽는다.

길 ③ — 운영체제 없는 기계에서 쓰려 한다

제1부 → 4장 → 제4부 → 18·20장 → 27장 → 29·30장 → 35장. 힙이 거절되는 자리(18장)와 기계의 등급(30장)이 이 길의 핵심이다.

길 ④ — 이 언어의 주장이 어디까지 참인지 알고 싶다

1장 → 14장 → 15장 → 12장 → 제10부 전체. 제10부의 각 장은 앞의 어느 장과 이어지는지 첫머리에 적어 두었다.

본문의 장치

장은 같은 모양으로 시작한다 — 이 장의 차례, 먼저 알아야 할 것, 그것을 꺼내 보는 문답, 이 장이 왜 이 자리에 있는가, 끝나면 무엇을 얻는가, 이 장에서 답할 질문. 본문에는 문답·흔한 오해·실제 사례·수학 상자·시연(예제와 실제 출력)이 나오고, 장 끝의 복습 정리가 요점을 한 번 모은다. 시연의 출력은 검증 스크립트가 남긴 것을 그대로 싣는다. 출력에 보이는 긴 영문 진단은 컴파일러가 실제로 낸 문장이다.

제1부 — 시작

이 부는 Lowent 로 무엇을 계산하는 법을 아직 가르치지 않는다. 대신 **이 언어와 처음 만나는 자리**를 깐다. 왜 이런 언어가 필요한지(1장), 도구를 짓고 돌리고 거절당하는 법(2장), 그리고 모든 코드가 따르는 겉모습의 규칙(3장)이다.

목표 (1장) — op 의 머리만 읽고 그 op 이 할 수 있는 일과 못 하는 일을 알게 한다는 목표와, 그것을 떠받치는 다섯 가지 생각을 세운다.

도구 (2장) — 같은 프로그램을 VM 과 네이티브로 돌려 맞대고, 계약(op 이 스스로 적는 약속)이 실행을 멈추는 모습과 진단 코드를 읽는 법을 익힌다.

겉모습 (3장) — 떨어진 마침표, `do … end` 블록, 전위 표기, 그리고 op 머리의 절 차례표. 이 표는 책 끝까지 쓰인다.

1 Lowent 는 무엇을 하려는 언어인가

이 장의 차례

1.1 머리만 읽고 알 수 있는가 19
1.2 다섯 가지 생각 21
1.3 한 뜻에 한 표기 22
1.4 이 언어의 처지 22
1.5 이 책을 읽는 법 22

이 장의 필요성과 맥락

새 언어의 첫 장이 문법으로 시작하면, 독자는 낱말을 외우다가 **그 낱말들이 왜 그렇게 생겼는지**를 놓친다. Lowent 의 문법은 대부분 한 가지 목표에서 나온다 — **op 의 머리만 읽어도 그 op 이 무엇을 할 수 있고 무엇을 못 하는지 알게 한다.** 그 목표를 먼저 세워 두면 뒤에 나오는 낯선 규칙들(마침표로 닫는 절, 순수를 적는 `fn`, 인자로 건네받는 권한)이 임의의 취향이 아니라 한 설계의 여러 얼굴로 보인다. 그래서 첫 자리에 목표와 이 책의 읽는 법을 놓는다.

이 장이 끝나면

Lowent 가 어떤 문제를 풀려고 만든 언어인지, 그 해법이 되는 다섯 가지 생각(순수의 표시·계약·효과와 권한·쓰레기 수거 없는 메모리·두 백엔드의 대조)이 무엇인지 알게 된다. 이 언어가 아직 실험 단계라는 사실과, 이 책의 예제가 어떻게 검증되는지, 어떤 장치로 읽으면 되는지도 여기서 정한다.

이 장에서 답할 질문

1. 머리에 적는 것이 그렇게 많으면 코드가 길어지지 않는가?
2. 문법이 그렇게 좁으면 새 기능은 어떻게 들어오는가?

1.1 머리만 읽고 알 수 있는가

프로그램을 고치는 사람이 가장 많이 하는 일은 **남이 쓴 함수를 부르기 전에 그 함수가 무엇을 하는지 알아내는 일**이다. 이름은 `parse_header` 인데 파일을 열지는 않는가? 메모리를 할당하는가? 실패하면 어떻게 알려 주는가? 스레드를 띄우는가? 대부분의 언어에서 이 질문의 답은 **본문**에 있다. 본문을 읽고, 본문이 부르는 함수의 본문을 읽고, 또 그 아래를 읽는다.

Lowent 는 그 답을 **머리**로 끌어올린다. 다음은 이 책에서 처음 만나는 op 의 머리다.

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

낱말을 모르더라도 몇 가지는 읽힌다. 이 op 은 `proc` 이다(순수하지 않다). `out` 이라는 이름으로 `io` 권한을 **받는다**. 돌려주는 값은 `u8` 이다. 그리고 `io` 라는 효과를 낸다. 반대로 적혀 있지 않은 것도 읽힌다 — 이 op 은 메모리를 할당하지 않고(`alloc` 이 없다), 파일 시스템 권한이 없으며, 스레드를 띄우지 않는다(`concurrent` 가 없다). **적혀 있지 않으면 할 수 없다.** 컴파일러가 그것을 검사한다.

실제로 돌아가는 작은 프로그램으로 보자.

```
examples/ch01/heads.low
```

```
module heads .
rem run: main

rem fn — 같은 입력이면 같은 답. 화면에도 파일에도 닿지 않는다
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 — 바깥에 흔적을 남길 수 있다. 무엇을 하는지(effects io)와 누가 허락했는지(cap i
o)가 머리에 보인다
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
```

실행 결과

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

- `module heads .` 는 이 파일의 이름표다. 모든 파일이 이 한 줄로 시작한다.
- `rem` 으로 시작하는 줄은 사람에게 하는 말(주석)이다. 컴파일러는 읽지 않는다.
- `fn area … do … end` 는 넓이를 셈하는 op 이다. `input w u64 .` 는 "`w` 라는 이름으로 부호 없는 64 비트 정수를 받는다" 이고, `output u64 .` 는 돌려줄 값의 타입이다. `requires le w 1000 .` 는 "`w` 는 1000 이하여야 한다" 는 약속이다. 곱이 넘치지 않게 하려는 것이다.
- `return mul w h .` 는 `w` 곱하기 `h` 를 돌려준다. 연산 이름이 앞에 오고 인자가 뒤에 온다. 문장은 떨어진 마침표로 끝난다.
- `main` 은 프로그램이 시작하는 op 이다. `let a u64 be area 3 4 .` 로 12 를 얻어 `a` 에 담고, `write_out out 1 "…"` 로 표준출력(1 번)에 한 줄을 쓴다. 이 줄이 가능한 까닭은 머리에 `cap io` 를 받았고 `effects io` 를 적었기 때문이다. 돌려준 12 는 운영체제에 건네는 종료 값이 된다.

이제 머리가 거짓말을 하게 해 보자. 순수하다고 적은 `fn` 이 몰래 화면에 쓰려 한다.

examples/ch01/mistake_hiddenio.low

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

rem ✘ fn 이라고 적고서 몰래 화면에 쓰려 한다
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
```

실행 결과

```
$ 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
```

컴파일러는 실행해 보지도 않고 거절한다. `fn` 이라는 한 낱말이 "이 op 은 바깥에 닿지 않는다" 는 약속이고, 본문이 그 약속을 어기면 번역이 멈춘다. 그래서 `fn` 을 부르는 사람은 본문을 열어 보지 않아도 된다. 고치려면 `proc area_loud … effects io .` 로 적어 사실대로 말하면 된다.

> **문. 머리에 적는 것이 그렇게 많으면 코드가 길어지지 않는가?**
>
> **답.** 길어진다. 대신 읽는 쪽이 짧아진다. 한 op 을 쓰는 일은 한 번이지만 읽는 일은 그 op 을 부르는 모든 자리에서, 고치는 모든 날에 일어난다. Lowent 는 그 비대칭에 걸었다. 그리고 머리의 절들은 모두 컴파일러가 검사하므로 주석처럼 낡지 않는다.

1.2 다섯 가지 생각

머리를 믿을 수 있게 만드는 장치는 다섯이다. 이 책의 제2부부터 제8부까지가 이 다섯을 하나씩 편다.

생각	머리에 적는 것	다루는 곳
순수와 비순수를 가른다	`fn` 또는 `proc`	2장
약속을 적고 검사받는다	`requires` · `ensures` · `errors`	14장
하는 일과 허락을 짝짓는다	`effects` · `input … cap …`	15·16장
수거기 없이 메모리를 지킨다	`region` · `owned` · `ref`	18–20장
두 가지로 실행해 맞대 본다	(머리가 아니라 도구)	31장

표 1.1 — Lowent 를 이루는 다섯 가지 생각

순수의 표시. `fn` 은 순수하다 — 같은 입력이면 같은 출력이고, 바깥에 흔적을 남기지 않는다. `proc` 은 그렇지 않을 수 있다. 둘 중 무엇인지는 **언제나** 적는다. 순수한 `fn` 이 출력을 하려 들면 컴파일러가 거절한다.

계약. op 은 호출자에게 요구하는 것(`requires`)과 돌려줄 때 보장하는 것(`ensures`)을 적는다. 값이 상수로 정해져 있으면 번역할 때 검사하고, 그렇지 않으면 실행할 때 진입과 반환에서 검사한다. 증명된 계약은 본문의 검사를 지운다 — 계약은 문서이면서 최적화의 근거다.

효과와 권한. op 이 세상에 남기는 자국을 `effects` 로 선언하고, 그 일을 할 권한을 `cap` 입력으로 **건네받는다**. Lowent 에는 주변 권한(ambient authority), 곧 어디서나 몰래 쓸 수 있는 전역 출력이나 전역 할당기가 없다. 권한이 인자 목록에 보이지 않으면 그 op 은 그 일을 못 한다.

수거기 없는 메모리. 쓰레기 수거기(GC) 없이 메모리를 안전하게 쓰려고, 값의 수명을 **영역**(region)에 묶고, 쓰는 자는 하나·읽는 자는 여럿이라는 차용 규칙을 검사한다. 운영체제가 없는 기계를 위해 자라지 않는 고정 창에서만 할당하는 길도 따로 있다.

두 백엔드의 대조. 컴파일러 `lowentc` 는 같은 중간 표현을 두 가지로 실행한다. 하나는 스스로 도는 가상 기계(VM)이고, 하나는 C 소스로 내보내 C 컴파일러가 기계어로 만드는 네이티브 코드다. 둘이 다른 답을 내면 컴파일러의 결함이다. 이 책의 예제는 전부 이 대조를 통과했다.

> **흔한 오해. Lowent 는 모든 것을 정적으로 증명하는 언어다**
>
> 아니다. 차용·영역·정수 확대 같은 것은 번역할 때 막지만, 증명되지 않은 자리의 경계 검사·넘침·계약 검사는 **실행할 때** 막는다. 섞어 쓰는 설계다. 무엇이 증명되었고 무엇이 아닌지는 제10부가 따로 정리한다 — 특히 50장은 **증명하지 않은 것**만 모은 장이다.

1.3 한 뜻에 한 표기

Lowent 의 이름에 들어 있는 **엔트로피**는 무질서의 척도다. 이 언어는 같은 뜻을 여러 가지로 적을 수 있는 자리를 줄이려 한다. 그래서 몇 가지 선택이 따라 나온다.

- 필드는 `field p x` 로만 읽는다. `p.x` 같은 붙임 점은 없다.
- 문장과 절은 홀로 선 마침표 `.` 로 닫고, 블록은 언제나 `do … end` 다.
- 머리의 절은 정해진 한 차례로만 적는다. 차례가 틀리면 거절되고, `--fmt` 가 고쳐 준다.
- 없앤 낱말(`loop`, `give`, `on` 따위)은 조용히 받아 주지 않고 `E-VOCAB-REMOVED` 로 거절한다.

한 뜻에 한 표기이면 사람과 도구가 같은 코드를 같은 모양으로 읽는다. 이 책의 예제가 모두 비슷하게 생긴 것은 저자의 취향이 아니라 언어가 그 모양만 허락하기 때문이다.

> **문. 문법이 그렇게 좁으면 새 기능은 어떻게 들어오는가?**
>
> **답.** 낱말을 늘리는 대신 **이미 있는 틀에 절을 하나 더하는** 식으로 들어온다. `pipe` 는 새 반복 문법이 아니라 `do … end` 블록 안에 줄마다 한 낱말을 적는 틀이고(24장), 병렬은 새 구문이 아니라 op 머리의 `parallel` 절이다(27장). 그래서 절의 차례표가 이 언어에서 가장 중요한 표 가운데 하나다(3장).

1.4 이 언어의 처지

Lowent 는 **실험 중인 언어**다. 버전은 1 이 되지 않았고, 문법과 표준 라이브러리는 아직 바뀐다. 이 책은 지금 저장소의 컴파일러가 실제로 받아들이는 문법으로 쓰였고, 컴파일러가 바뀌면 책도 따라 바뀐다. 책의 판 번호가 따로 있는 이유다.

컴파일러는 C23 으로 쓰였고 외부 의존은 벤더링한 `proven_c_lib` 하나다. 네이티브 코드는 C 로 내보내 시스템의 C 컴파일러로 짓는다. 그래서 C 컴파일러가 있는 곳이면 대개 돌아가고, 운영체제가 없는 마이크로컨트롤러도 대상에 들어 있다(30장).

> **실제 사례. 이 책의 예제는 어떻게 검증되나**
>
> 책에 실린 모든 `.low` 파일은 `docs/manual/examples/` 에 있고, 파일마다 첫머리에 지시가 한 줄 붙어 있다. `rem run: …` 이면 VM 과 네이티브로 둘 다 실행해 출력이 같아야 하고, `rem expect: E-…` 이면 컴파일러가 바로 그 진단으로 거절해야 하며, 지시가 없으면 `--check` 를 통과해야 한다. 지면의 실행 결과는 사람이 옮겨 적지 않았다 — 검증 스크립트가 남긴 출력을 그대로 싣는다.

1.5 이 책을 읽는 법

이 책은 두 부류의 독자를 함께 생각하며 썼다.

프로그래밍이 처음인 독자는 제1부부터 차례대로 읽는다. 문법 하나마다 “무엇을 하는가” 와 함께 “왜 이렇게 생겼는가” 를 적었고, 예제 뒤에는 그 줄들이 무엇을 하는지 풀이가 따라온다. 제2장부터 제37장

까지 장 끝의 “흔한 실수” 는 처음 배우는 사람이 실제로 부딪히는 오류와 그 진단을 그대로 보여 준다. 오류 메시지를 두려워하지 않아도 된다. 이 언어에서 진단은 “틀렸다” 가 아니라 “여기가 약속과 다르다” 는 안내다. 처음 만나는 말은 아래 표로 먼저 익혀 둔다.

다른 언어를 써 본 독자는 예제 코드와 각 장의 “이 장의 문법 한눈에” 표만 따라가도 된다. 예제와 그 표가 그 장의 문법을 함께 보여 준다. C 를 알면 제5부와 제8부가 쉽고, Rust 를 알면 제5부의 차용 규칙이 익숙할 것이다. 익숙한 언어에서 가져온 생각이 여기서 틀리는 자리는 “흔한 오해” 상자에 모았다.

말	뜻
소스 · 모듈	사람이 쓰는 `.low` 파일 하나 · 그 파일이 이루는 이름 붙은 단위(21장)
컴파일러 · 번역	소스를 읽어 검사하고 실행할 수 있는 모양으로 바꾸는 프로그램(`lowentc`) · 그 일
진단	컴파일러가 알려 주는 오류(`E-`)·경고(`W-`)·알림(`N-`). 이름이 곧 찾아볼 열쇠다(부록 B)
값 · 타입	수나 글자 같은 자료 하나 · 그 값이 어떤 종류이고 몇 비트인지(`u64` 는 부호 없는 64비트 정수)
지역 이름	값에 붙인 이름. 바꾸지 않으면 `let`, 바꾸면 `var`(6장)
op	이름을 붙인 일 한 덩어리. 다른 언어의 함수다. 순수하면 `fn`, 아니면 `proc`(5장)
효과 · 권한	op 이 바깥에 남기는 자국(`effects io`) · 그 일을 해도 된다는 허락(`cap io`)(15·16장)
계약	op 이 받기 전에 요구하고 돌려줄 때 보장하는 조건(`requires` · `ensures`)(14장)

표 1.2 — 처음 만나는 말

각 장은 같은 모양으로 시작한다. 먼저 그 장이 기대는 앞 장을 적고(**먼저 알아야 할 것**), 그것을 한 번 꺼내 보는 문답을 두고, 이 장이 왜 이 자리에 있는지와 끝나면 무엇을 얻는지를 적은 뒤, 장에서 답할 질문 목록을 보인다. 본문 중간의 장치는 다음과 같다.

장치	쓰임
문답	읽다가 떠오를 법한 질문과 그 답
흔한 오해	그럴듯하지만 틀린 생각과, 왜 틀렸는지
실제 사례	이 언어나 표준 라이브러리에서 실제로 있었던 일
수학 상자	건너뛰어도 본문이 이어지는 형식적 설명
시연	예제 파일과 검증 스크립트가 남긴 실제 출력

표 1.3 — 본문에서 만나는 장치

처음 읽는다면 제1부부터 제4부까지는 차례대로 읽기를 권한다. 그 뒤로는 필요한 부로 건너뛰어도 된다. 표준 라이브러리를 찾는다면 제9부로, 이 언어의 주장이 어디까지 참인지 궁금하다면 제10부로 가면 된다.

2 첫 프로그램 — 짓고, 돌리고, 거절당하기

이 장의 차례

2.1 도구 짓기 25
2.2 hello, entropy 25
2.3 두 가지로 돌린다 26
2.4 계약이 멈추는 모습 27
2.5 거절당하기 27
2.6 모양을 고쳐 주는 `--fmt` 28
2.7 인자와 권한 29
2.8 흔한 실수 29
2.9 이 장의 문법 한눈에 32

먼저 알아야 할 것

1장 Lowent 는 무엇을 하려는 언어인가 · 머리만 읽고 op 이 할 일을 안다는 목표

돌아보기

1장은 `proc main input out cap io . output u8 . effects io .` 라는 머리에서 무엇을 읽어 냈는가? 그리고 **적혀 있지 않은 것**에서는 무엇을 읽어 냈는가?

> **답.** 순수하지 않은 op 이고(`proc`), `io` 권한을 `out` 이라는 이름으로 받으며, `u8` 을 돌려주고, `io` 효과를 낸다는 것을 읽었다. 적혀 있지 않은 것에서는 이 op 이 할당하지 않고, 파일 시스템 권한이 없고, 스레드를 띄우지 않는다는 것을 읽었다. 이 장에서 그 머리를 가진 프로그램을 실제로 짓고 돌린다.

이 장의 필요성과 맥락

언어를 배우는 첫 걸음은 **손에 쥔 도구가 무엇을 해 주는지**를 아는 것이다. Lowent 의 도구 `lowentc` 는 기본 모드가 없고, 같은 프로그램을 두 가지 방식(VM 과 네이티브)으로 돌리며, 틀린 프로그램에는 코드가 붙은 진단을 낸다. 이 세 가지를 문법보다 먼저 몸에 익혀 두면, 뒤의 장에서 새 규칙을 만날 때마다 직접 돌려 보고 직접 거절당해 볼 수 있다. 그래서 문법 설명(3장)보다 이 장이 앞선다.

이 장이 끝나면

`lowentc` 를 짓고, 가장 작은 프로그램을 VM 으로 한 번·네이티브로 한 번 돌리게 된다. 돌려주는 값이 종료 코드가 되는 이유, 계약이 실행을 멈추는 모습, `test` 블록을 돌리는 법을 보게 된다. 컴파일러가 거절할 때 내는 진단을 읽는 법과, `--fmt` 가 모양을 고쳐 주는 범위도 알게 된다.

이 장에서 답할 질문

1. 왜 기본 모드가 없는가? `lowentc hello.low` 만 쳐도 알아서 짓게 할 수 있지 않은가?
2. 진단 코드는 외워야 하는가?

2.1 도구 짓기

컴파일러의 이름은 `lowentc` 다. 저장소를 받은 뒤 `impl/` 에서 짓는다. 필요한 것은 C23 을 아는 C 컴파일러(gcc 나 clang)와 `make` 뿐이다.

```
cd impl
make            # build/lowentc
make check      # 단위 시험과 표준 라이브러리 검사
```

`make` 가 끝나면 `impl/build/lowentc` 가 생긴다. 이 책의 명령은 그것이 경로에 있다고 치고 `lowentc` 로 적는다.

모드	하는 일
`--check`	검사만 한다(계약·효과·소유·권한). 아무것도 내보내지 않는다
`--run OP 인자…`	VM 으로 op 하나를 돌린다. 슬라이스 인자는 `[3,4,8]` 로 준다
`--emit-c`	네이티브 C 소스를 표준출력으로 낸다
`--test`	`test` 블록을 돌린다
`--fmt`	정규형으로 다시 찍는다
`--diag-json`	진단을 JSON 한 줄씩 낸다(도구와 AI 가 읽는다)

표 2.1 — 자주 쓰는 `lowentc` 모드

> **문. 왜 기본 모드가 없는가? `lowentc hello.low` 만 쳐도 알아서 짓게 할 수 있지 않은가?**
>
> **답.** 할 수 있지만 그러면 **그 명령이 무엇을 했는지**가 명령 줄에 안 보인다. 검사만 했는지, 실행했는지, 파일을 썼는지를 사람이 기억해야 한다. 1장의 "한 뜻에 한 표기" 가 도구에도 적용된 것이다. 짧게 쓰고 싶으면 `lowentc check`, `lowentc run` 같은 서브커맨드 이름이 같은 일을 한다.

2.2 hello, entropy

가장 작은 출력 프로그램이다.

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
```

실행 결과

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

첫 줄 `module hello .` 는 모든 파일이 여는 모듈 선언이다. 둘째 줄의 `rem` 은 줄 주석이고, 이 책에서는 검증 스크립트에게 무엇을 할지 알려 주는 데 쓴다.

op 의 머리를 한 절씩 읽는다.

- `proc main` — 이름이 `main` 인 비순수 op. 출력은 바깥에 흔적을 남기므로 `fn` 이 될 수 없다.

- `input out cap io .` — `io` 권한을 `out` 이라는 이름으로 받는다. 표준출력에 쓰려면 이 권한이 있어야 한다.
- `output u8 .` — `u8` 을 돌려준다.
- `effects io .` — 입출력 효과를 낸다고 선언한다.

본문은 `do` 와 `end` 사이다. `write_out out 1 "…"` 는 권한 `out` 을 첫 인자로 받아 파일 기술자 `1`(표준출력)에 바이트를 쓰고, **쓴 바이트 수**를 돌려준다. `hello, entropy!\n` 는 16 바이트이므로 16 이 나온다. 그것을 `narrow u8` 로 `u8` 에 맞게 좁혀 돌려준다. 좁히기는 값이 들어가지 않으면 실행 중에 멈추는 검사를 동반한다(4장).

실행 결과의 마지막 줄 `main() = 16` 은 VM 이 op 의 반환값을 보여 주는 줄이다.

2.3 두 가지로 돌린다

`--run` 은 컴파일러 안의 가상 기계로 돌린 것이다. 같은 프로그램을 네이티브로 돌리려면 C 로 내보내고 C 컴파일러로 짓는다.

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

네이티브 실행 파일은 첫 인자로 돌릴 op 의 이름을 받는다. `main` 을 돌리면 화면에는 같은 한 줄이 찍히고, 반환값 16 은 **프로세스의 종료 코드**로 나온다(`echo $?` 로 본다). `main` 의 `output` 이 `u8` 인 까닭이다 — 종료 코드는 한 바이트다.

> **실제 사례. 두 결과가 다르면 누구의 잘못인가**
>
> VM 과 네이티브는 같은 중간 표현에서 출발한다. 둘이 다른 출력을 내면 프로그램이 아니라 **컴파일러**의 결함이다. 개발 저장소는 수백 개의 op 을 경계값 인자로 두 백엔드에서 돌려 바이트 단위로 맞대고, 이 책의 검증 스크립트도 `rem run:` 이 붙은 모든 예제를 두 번 돌려 비교한다. 이 책을 쓰는 동안에도 한 가지 어긋남이 드러났다 — 프로그램 인자를 읽는 `arg` 가 VM 과 네이티브에서 번호를 다르게 센다. 그래서 인자를 읽는 예제는 이 판에서 검사만 하고 실행 결과를 싣지 않는다.

슬라이스 인자는 대괄호로 준다. 다음 op 은 바이트 슬라이스의 평균을 구한다.

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
```

실행 결과

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

`requires gt (len xs) 0 .` 는 **계약**이다. 호출자는 빈 슬라이스를 주지 않는다고 약속한다. 그 약속이 있으니 본문의 `div total (len xs)` 가 0 으로 나눌 걱정을 하지 않는다. 결과 아래의 `arg0 (written) = [3,4,8]` 은 VM 이 슬라이스 인자의 마지막 모습을 보여 주는 줄이다.

2.4 계약이 멈추는 모습

계약을 어기면 어떻게 되는가. 다음 op 은 100 이하만 받는다.

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
```

실행 결과

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

`lowentc --run twice twice.low 200` 처럼 약속을 어긴 값을 주면, VM 은 본문에 들어가기 전에 멈추고 `E-VM-CONTRACT: requires violated at entry` 를 낸다. 인자가 상수로 적힌 호출(`twice 250`)이면 실행할 필요도 없이 `--check` 가 `E-CONTRACT-IMPOSSIBLE` 로 거절한다. 계약은 주석이 아니다.

같은 파일의 `test twice_works` 블록은 시험이다. `expect` 가 단언이고, `--test` 가 모든 `test` 블록을 돌린다. 결과의 마지막 두 줄이 그 출력이다. `--check` 는 시험을 돌리지 않는다 — 검사가 통과했다고 시험이 통과한 것은 아니므로, 시험이 있는 파일을 `--check` 하면 컴파일러가 `W-TEST-NOT-RUN` 경고로 그 사실을 일러 준다.

> **흔한 오해. `--check` 가 조용하면 프로그램이 옳다**
>
> `--check` 가 보증하는 것은 **머리에 적은 약속과 본문이 어긋나지 않는다**는 것까지다. 적지 않은 약속은 검사할 수 없고, 시험은 돌리지 않았으며, 실행 중에만 알 수 있는 계약 위반은 실행해야 드러난다. "검사 통과" 와 "시험 통과" 와 "옳다" 는 세 가지 다른 말이다.

2.5 거절당하기

Lowent 를 배우는 시간의 상당 부분은 컴파일러에게 거절당하는 시간이다. 거절은 벌이 아니라 대화다. 진단마다 안정된 코드가 붙어 있고, 그 코드는 판이 바뀌어도 같은 뜻을 지킨다.

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
```

실행 결과

```
$ 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
```

진단 한 줄은 `파일:줄:칸 코드: 설명` 의 꼴이다. 이 예에서 `fn shout` 은 순수하다고 선언했는데 본문이 `write_out` 으로 입출력을 한다. 컴파일러는 `E-EFFECT-CALC` 로 거절하고, 고치는 길 둘을 말한다 — `proc` 으로 바꾸고 `effects` 를 적거나, 효과를 없애거나.

`--diag-json` 을 붙이면 같은 진단이 JSON 한 줄로 나온다. 규칙 코드(`rule`), 위치(`span`), 고치는 방법의 이름(`repair`)이 칸으로 나뉘어 있어서, 편집기나 AI 가 설명 문장을 뜯지 않고도 무엇이 틀렸는지 안다.

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

> **문. 진단 코드는 외워야 하는가?**
>
> **답.** 외울 필요는 없다. 코드는 검색하고 참조하라고 붙인 이름이다. 이 책의 부록에 자주 만나는 진단을 모았고, 각 장은 그 장의 규칙을 어길 때 나오는 코드를 본문에서 보여 준다.

2.6 모양을 고쳐 주는 `--fmt`

op 머리의 절은 정해진 한 차례로 적는다. 입력이 출력보다 먼저다. 차례가 틀리면 거절된다.

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
```

실행 결과

```
$ 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)
```

진단의 긴 설명이 곧 차례표다. 전부는 3장에서 다루고, 지금은 한 가지만 기억한다: `--fmt` 는 **입력이 아닌 절**(출력·효과·계약 따위)을 제자리로 옮겨 준다. 입력끼리의 차례는 옮기지 않는다. 입력의 차례는 부르는 쪽 인자의 차례이기도 해서, 옮기면 모든 호출 자리가 함께 바뀌어야 하기 때문이다.

`--fmt` 가 찍는 정규형은 절마다 한 줄이고 괄호를 모두 드러낸다. 사람이 그 모양으로 쓸 필요는 없다. 정규형은 **두 코드가 같은 뜻인지**를 기계가 비교하는 기준이다.

2.7 인자와 권한

프로그램 인자를 읽는 것도 권한이다. 다음 프로그램은 첫 인자를 받아 인사한다.

examples/ch02/greet.low

```
module greet .
rem (인자는 네이티브와 VM 의 번호가 달라 여기서는 검사만 한다)

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
```

실행 결과

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

`input a cap args .` 로 인자 권한을 받고, `arg a 0` 이 첫 인자를 읽는다. 인자가 없을 수도 있으니 결과는 `option slice u8` 이다. `guard is_some who . else …` 는 값이 없으면 그 자리에서 떠나고, 그 아래에서는 값이 있다고 믿어도 된다(11장). 권한 입력이 데이터 입력보다 앞에 오고, 권한 둘은 적은 차례대로 인자 자리를 차지한다.

`effects io .` 한 줄이 두 권한을 모두 덮는 것에 주목한다. `args` 는 효과 없이 읽을 수 있는 권한이다. 권한과 효과가 어떻게 짝지어지는지는 16장이 표로 정리한다.

2.8 흔한 실수

첫 프로그램에서 가장 자주 만나는 거절과 멈춤이다. 진단 한 줄의 코드를 보고 여기로 돌아오면 된다.

> **반례. `do` 를 열고 `end` 로 닫지 않는다**
>
> **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 ✘ 몸을 연 `do` 에 짝이 되는 `end` 가 없다
```

실행 결과

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

몸은 `do` 로 열고 `end` 로 닫는다. 파일이 끝날 때까지 닫는 말이 없으면 컴파일러는 어디까지가 몸인지 알 수 없어 `E-BLOCK-UNCLOSED` 를 낸다. 진단의 `5:1` 은 짝을 잃은 `do` 의 자리다 — 몸이 여럿 겹쳐 있으면 그 줄부터 짝을 맞춰 본다. 들여쓰기를 맞춰 두면 짝이 눈에 보인다.

반례. 출력하면서 권한을 받지 않는다

examples/ch02/mistake_nocap.low

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

rem ✘ `io` 를 한다고 적었지만 권한(`cap io`)을 받지 않았다
proc main output u8 . effects io .
do
  let n u64 be write_out 1 "hi\n" .
  return narrow u8 n .
end
```

실행 결과

```
$ 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 .` 는 "입출력을 한다" 는 **선언**이고, 실제로 할 수 있게 해 주는 것은 입력으로 받는 **권한**(`input out cap io .`)이다. 권한 없이 효과만 적으면 `E-EFFECT-NO-CAP` 이다. 입출력은 아무 데서나 꺼내 쓰는 힘이 아니라 `main` 이 시작할 때 건네받아 필요한 op 에게 넘겨주는 권리다(16장). 고치는 법: 머리에 `input out cap io .` 를 더하고, `write_out` 의 첫 인자로 `out` 을 준다.

반례. 종료 코드에 255 보다 큰 수를 돌려준다

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 바이트를 썼다 --- 종료 코드는 한 바이트(0 … 255)라 들어가지 않는다
  return narrow u8 n .
end
```

실행 결과

```
$ 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)
```

번역은 통과하지만 실행 중에 멈춘다. `main` 의 반환값은 운영체제에 넘기는 **종료 코드**라서 한 바이트(`u8`)이고, 256 은 `narrow u8` 에 들어가지 않는다. Lowent 는 값을 조용히 잘라 0 을 돌려주지 않고 그 자리에서 멈춘다. 잘린 종료 코드는 "성공" 으로 읽혀 버그를 숨기기 때문이다. 쓴 바이트 수가 알고 싶은 것이라면 출력하고, 종료 코드로는 성공(0)·실패(0 아닌 수)만 돌려준다.

반례. 돌릴 op 의 이름을 잘못 적는다

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
```

실행 결과

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

`--run` 뒤의 이름은 파일 안의 op 이름과 글자 하나까지 같아야 한다. `mian` 은 없으므로 VM 이 `E-VM-UNDEF: no such op` 로 멈춘다. 네이티브 실행 파일도 첫 인자로 받은 이름의 op 이 없으면 비영으로 끝난다. 무엇을 돌릴지 명령 줄에 보이게 한 대가다 — 대신 같은 파일 안의 어느 op 이든 따로 돌려 볼 수 있다.

2.9 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`module hello .`	이 파일의 모듈 이름	다른 파일이 `use hello .` 로 부를 이름
`rem …`	줄 주석(줄 끝까지)	기호 없이 낱말로 — 이 책에서는 검증 지시도 싣는다
`proc main input out cap io . output u8 . effects io .`	프로그램의 시작점	받는 권한·돌려주는 종료 코드·하는 일이 머리에 다 보인다
`do … end`	몸	여는 말과 닫는 말이 짝을 이룬다
`write_out out 1 "…"`	표준출력(1)에 쓰고 쓴 바이트 수를 돌려준다	권한 `out` 이 첫 인자 — 권한 없이는 못 쓴다
`narrow u8 n`	`u8` 로 좁힌다(안 들어가면 멈춘다)	값이 조용히 바뀌지 않게
`requires c . • test t do … end • expect c .`	계약 · 시험 블록 · 시험 속 단언	약속은 검사되고, 시험은 따로 돈다
`lowentc --check f.low`	검사만 한다	무엇을 했는지 명령 줄에 보이게 — 기본 모드가 없다
`lowentc --run op f.low 인자…`	VM 으로 op 하나를 돌린다	파일 안의 어느 op 이든 따로 돌려 본다
`lowentc --emit-c f.low > f.c`	네이티브용 C 를 낸다	VM 과 같은 답인지 맞대 볼 수 있게
`lowentc --test • --fmt`	시험을 돌린다 · 정규형으로 다시 찍는다	검사와 시험과 모양은 서로 다른 일
`lowentc --hw auto f.low`	기계의 암호 명령과 폭을 담는다(AES-NI·캐리 없는 곱셈·SSE2/AVX2 레인)	빌드가 **담을 것**을 정하고, 담은 것이 둘 이상이면 시작할 때 한 번 고른다 — 답은 어느 쪽이든 같다

표 2.2 — 첫 프로그램의 모양과 도구 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`lowentc` 는 모드를 반드시 고른다. `--run` 은 VM 으로, `--emit-c` 와 C 컴파일러는 네이티브로 같은 프로그램을 돌리고, 둘은 같은 답을 내야 한다. `main` 의 반환값은 종료 코드다. 계약은 실행 전이나 진입에서 멈추고, `test` 는 `--test` 로만 돈다. 거절에는 안정된 코드가 붙고, 절의 차례는 `--fmt` 가 입력 밖의 절에 한해 고친다.

3 겉모습 — 마침표, 블록, 절의 차례

이 장의 차례

3.1 이름이 먼저, 인자가 뒤 34
3.2 떨어진 마침표가 닫는다 34
3.3 블록은 언제나 `do … end` 36
3.4 주석과 리터럴 36
3.5 이름 39
3.6 낱말은 예산이다 40
3.7 `expr` 섬의 경계 40
3.8 op 머리의 절 차례 40
3.9 흔한 실수 41
3.10 이 장의 문법 한눈에 45

먼저 알아야 할 것

1장 Lowent 는 무엇을 하려는 언어인가 · 한 뜻에 한 표기
2장 첫 프로그램 · `--check` 의 진단과 `--fmt`

돌아보기

2장에서 `fn area output u64 . input w u64 . …` 는 무엇으로 거절되었고, `--fmt` 는 그것을 어디까지 고쳐 주는가?

> **답.** `E-CLAUSE-ORDER` 로 거절되었다. 데이터 입력이 `output` 뒤에 왔기 때문이다. `--fmt` 는 입력이 아닌 절(출력·효과·계약)을 제자리로 옮기지만 입력끼리의 차례는 건드리지 않는다. 입력의 차례는 곧 부르는 쪽 인자의 차례이기 때문이다. 이 장은 그 차례표 전체와, 그 표를 받치는 더 작은 규칙들(마침표·괄호·블록)을 다룬다.

이 장의 필요성과 맥락

Lowent 코드는 처음 보면 낯설다. 괄호 대신 마침표가 많고, 연산자 대신 낱말이 앞에 온다. 이 낯섦을 낱말 하나하나의 예외로 외우면 오래 걸린다. 그러나 겉모습의 규칙은 몇 개 되지 않고, 모두 1장의 "한 뜻에 한 표기" 에서 나온다. 뒤의 모든 장이 이 겉모습 위에서 쓰이므로, 값과 흐름(제2부)으로 들어가기 전에 한 번에 정리해 둔다.

이 장이 끝나면

떨어진 마침표가 폼을 닫고 개행은 아무것도 닫지 않는다는 것, 이름이 먼저 오는 전위 표기와 중위 표기를 허락하는 `expr` 섬의 경계를 알게 된다. 주석과 리터럴의 규칙, 이름에 점이 없고 가림이 금지된다는 것, 모든 블록이 `do … end` 라는 것을 익힌다. 마지막으로 op 머리의 절 차례표 전체와 그 차례가 왜 그렇게 놓였는지를 이해하게 된다.

이 장에서 답할 질문

1. 산술이 길어지면 괄호가 겹겹이 쌓여 읽기 어렵지 않은가?
2. 모듈 이름과 op 이름이 같아도 되는가?

3.1 이름이 먼저, 인자가 뒤

Lowent 의 식은 **전위 표기**다. 연산의 이름이 먼저 오고 인자가 뒤에 온다. `add a b` 는 `a` 와 `b` 를 더하고, 폼 안에 폼이 오면 괄호로 감싼다.

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

전위 표기에는 우선순위가 없다. `1 + 2 * 3` 을 읽는 사람은 곱셈이 먼저라는 규칙을 **외워서** 알지만, `add 1 (mul 2 3)` 은 괄호가 이미 말해 준다. 이 원칙은 사용자가 만든 op 에도 똑같다. `write_out out 1 "hi"` 도, `field p x` 도, `mean xs` 도 모두 이름이 먼저다.

문. 산술이 길어지면 괄호가 겹겹이 쌓여 읽기 어렵지 않은가?

답. 그래서 `expr` 섬이 있다. `expr` 로 시작하는 자리 안에서는 사칙과 비교, `and`·`or` 를 중위로 쓸 수 있다. 섬 안의 식은 전위와 **완전히 같은 뜻**으로 번역되며 실행 비용은 없다. 섬은 이 장의 뒤에서 경계를 보고, 8장에서 자세히 다룬다.

3.2 떨어진 마침표가 닫는다

문장과 절의 끝은 **떨어져 있는** 마침표 `.` 로 닫는다. 이것을 폼을 닫는 **닫개**라 부른다. 개행은 닫개가 아니다 — 스페이스와 똑같은 공백이다. 그러니 줄을 어디서 나누어도 뜻이 같다.

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
```

실행 결과

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

`poly` 의 `return` 은 두 줄에 걸쳐 있지만 한 폼이다. 폼은 둘째 줄 끝의 마침표에서 끝난다. 줄을 이어 쓰는 표시(C 의 행끝 역슬래시, 줄 끝의 쉼표, 들여쓰기 규칙)가 따로 없는 이유다. 같은 파일의 `poly_expr` 은 같은 계산을 `expr` 섬으로 적었고, 두 op 은 같은 답을 낸다.

`note WHY … WHY` 는 여러 줄 주석이다. `note` 다음에 적은 낱말이 홀로 서는 줄에서 주석이 끝난다. `rem` 은 줄 주석이다. 이 언어에는 `//` 나 `/* */` 같은 기호 주석이 없다 — 주석도 낱말로 연다.

마침표의 개수는 일종의 **검사합**이다. 열린 폼과 닫는 표시가 맞지 않으면 컴파일러가 알린다. 괄호 하나를 빠뜨리면 이렇게 된다.

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
```

실행 결과

```
$ 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 ')'
```

첫 진단은 괄호가 블록의 `end` 를 넘어 새어 나가려 했다는 뜻이다. 괄호는 블록 경계를 넘지 못한다. 그래서 괄호 하나를 잘못 닫은 실수가 파일의 나머지를 조용히 삼키는 일 없이, 그 블록 안에서 멈춘다.

흔한 오해. 세미콜론도 문장을 닫는다

> 한때는 그랬다. 처리기가 `;` 를 만나면 글자 그대로 마침표를 냈다. 같은 뜻의 철자가 둘이면 읽는 사람이 둘을 다 알아야 하므로 없앴다. 지금 `;` 는 `E-VOCAB-REMOVED` 로 거절된다.

examples/ch03/semi.low

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

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

실행 결과

```
$ 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 블록은 언제나 do … end

여러 문장을 묶는 자리는 모두 `do` 로 열고 `end` 로 닫는다. op 의 본문, `if`·`while`·`for` 의 몸, `match` 의 갈래, 그리고 `struct`·`enum`·`trait`·`actor` 같은 선언의 몸이 모두 그렇다.

```
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` 는 괄호 한 쌍과 같다. `end` 는 **자기 `do` 만** 닫고, 그 바깥은 건드리지 않는다. 그래서 알아 둘 규칙은 둘뿐이다.

- **블록을 몸으로 갖는 구문**은 `end` 에서 끝난다. op 선언, `struct`·`enum`, `if`·`while`·`for`·`match` 가 그렇다. 위의 예처럼 `end` 뒤에 마침표를 찍지 않는다 — 찍으면 닫을 것이 없어 `E-DOT-STRAY` 로 거절된다.
- **블록을 값으로 쓰는 문장**은 여느 문장처럼 자기 마침표로 끝난다. `make` 로 구조체 값을 만들어 `let` 에 넣는 경우가 대표적이다: `let p point be make point do x 1 . y 2 . end .` — 마지막 마침표는 `let` 의 것이다. 빠뜨리면 `E-DOT-MISSING`.

C 에 견주면 `if (c) { … }` 뒤에는 `;` 가 없고 `p = (struct point){ 1, 2 };` 뒤에는 있는 것과 같다.

```
while le i n . do  …  end              몸-블록: while 문이 end 에서 끝난다
└──────── while 문 ────────┘

let p point be make point do x 1 . end .   값-블록: 블록은 make 의 것, 문장은 자기 . 으로 끝난다
              └──────── make 값 ────────┘ |
└──────────────── let 문 ─────────────────┘
```

블록 안의 문장도 저마다 자기 마침표로 끝나야 한다. `end` 가 안에 열린 문장을 대신 닫아 주지 않는다. 머리 없이 `do … end` 만 따로 적는 것도 안 된다(`E-BLOCK-NOHEAD`) — 블록은 언제나 그것을 여는 머리가 있다. 선언 블록을 `do` 없이 줄바꿈만으로 열면 `E-STMT-NODO` 로 거절되고 `--fmt` 가 `do` 를 넣어 준다. op 머리의 절도 같다. `input n u64 .`·`output u64 .` 처럼 절마다 자기 마침표로 끝나고, 다음 절 낱말이 앞 절을 대신 닫아 주지 않는다 — 빠뜨리면 `E-DOT-MISSING`.

3.4 주석과 리터럴

갈래	예
정수(십진·십육진·이진)	`42` · `0xFF` · `0b1010` · `1_000_000`
부동소수	`3.14` · `6e-3` · `0x1.8p3`
문자열(바이트)	`"hello\n"`
문자열(UTF-16 · 코드포인트)	`u"…"` · `U"…"`

참거짓 · 부재	`true` · `false` · `none`

표 3.1 — 리터럴을 적는 법

정수 리터럴은 그 자체로는 타입이 없고 쓰이는 자리가 타입을 정한다. 앞자리 0 은 팔진이 아니다 (`0755` 는 755 다). 리터럴이 자리의 타입에 들어가지 않으면 조용히 잘리지 않고 거절된다.

examples/ch03/lit300.low

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

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

실행 결과

```
$ 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 는 300 을 `u8` 에 넣으면 44 로 자른다. 그러면 소스에 적힌 값과 실제 값이 달라진다. Lowent 는 그것을 번역할 때 막는다. 줄이는 일은 `narrow` 계열의 이름 붙은 연산으로 **적어서** 한다 (4장).

문자열의 이스케이프는 닫힌 집합이다. `\n·\t·\\·\"` 같은 익숙한 것과 `\xNN`(정확히 두 자리), `\uXXXX`(네 자리), `\UXXXXXXXX`(여덟 자리)가 있고, 그 밖의 이스케이프는 `E-STR-ESCAPE` 로 거절된다. C 의 `\x` 는 16진 숫자를 끝없이 먹어서 `"\x41e"` 의 뜻이 옆 글자에 따라 달라지지만, Lowent 의 `\x41e` 는 언제나 `A` 다음 `e` 다. 문자열은 따로 된 타입이 아니라 바이트 슬라이스(`slice u8`)다.

적는 것	뜻	적는 것	뜻
`\\`	역슬래시	`\v`	세로 탭
`\"`	큰따옴표	`\0`	영 바이트
`\'`	작은따옴표	`\xNN`	십육진 두 자리 — 임의의 한 바이트
`\a`	경보	`\uXXXX`	코드포인트, 네 자리
`\b`	한 칸 뒤로	`\UXXXXXXXX`	코드포인트, 여덟 자리
`\f`	쪽 넘김	`\n`	줄 바꿈
`\r`	줄 처음으로	`\t`	가로 탭

표 3.2 — 문자열 이스케이프 — 이 열넷뿐이다

`\uXXXX` · `\UXXXXXXXX` 는 코드포인트를 적고, 그것이 몇 조각으로 실릴지는 접두사가 정한다 — `"\U0001F600"` 은 바이트 4 개, `u"…"` 은 UTF-16 코드 유닛 2 개(서로게이트 쌍), `U"…"` 은 코드포인트 1 개다. 서로게이트 자리(D800 … DFFF)와 10FFFF 초과는 거절된다. 팔진 이스케이프는 없다 — 팔진 표기가 언어에 아예 없으므로 리터럴에만 되살리면 그 자리가 유일한 예외가 된다.

표의 모든 모양을 한 파일에서 써 보면 이렇다.

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 정수 — 이진 · 자리 구분 _ · 십육진. 앞자리 0 은 팔진이 아니다
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 부동소수 — 지수 e · 십육진 부동소수 p(0x1.8 은 1.5, p1 은 곱하기 2)
fn floats output f64 . do
  let e f64 be 1.5e3 .
  let h f64 be 0x1.8p1 .
  return add e h .
end

rem 문자 — 'A' 는 바이트 하나, u'가' 는 UTF-16 코드 유닛 하나, U'😀' 는 코드포인트 하나
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 문자열 — \x41 은 정확히 두 자리라 "\x41e" 는 두 바이트, u"가나" 는 코드 유닛 둘
fn texts output u64 . do
  let x u64 be len "\x41e" .
  let w u64 be len u"가나" .
  rem text 태그 … 태그 — 여러 줄을 그대로. 본문의 \n 은 풀지 않는다
  let h u64 be len text DOC
line one
a\nb
DOC .
  return add (mul x 100) (add (mul w 10) h) .
end
```

실행 결과

```
$ 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` 는 10(`0b1010`) + 1000000(`1_000_000`) + 65535(`0xFF_FF`) + 755(`0755`) 이다. 밑줄은 자릿수 **사이**에만 오고 값에 아무 영향이 없다. 긴 수를 읽기 쉽게 하려는 것이다. `0755` 가 493 이 아니라 755 인 것은 C 의 함정을 없애려는 선택이다.
- `floats` 는 1500(`1.5e3`) + 3(`0x1.8p1` — 1.5×2^{1}) 이다. 십육진 부동소수는 비트를 정확히 적어야 할 때 쓴다. 십진 `0.1` 은 이진으로 정확히 적을 수 없지만 `0x1.8p1` 은 정확히 3 이다.
- `chars` 는 65(`'A'`) + 44032(`u'가'`) + 128512(`U'😀'`) 이다. 접두사가 원소의 폭을 정한다. `'가'` 처럼 한 바이트에 들어가지 않는 글자를 접두사 없이 적으면 `E-CHAR-WIDTH` 로 거절된다.
- `texts` 에서 `"\x41e"` 는 2 바이트, `u"가나"` 는 코드 유닛 2 개, `text DOC … DOC` 는 13 바이트다. 여러 줄 문자열(heredoc)은 `text` 뒤의 태그부터 같은 태그가 홀로 선 줄까지를 **적힌 그대로** 담는다. 본문의 `\n` 은 줄바꿈이 아니라 두 글자이고, 마지막 줄 뒤에는 줄바꿈이 붙지 않는다. 긴 설명문이나 시험용 입력을 이스케이프 없이 붙여 넣으려는 장치다.
- 맨 위의 `note END … END` 는 여러 줄 주석이다. `rem` 을 줄마다 붙이지 않고 한 덩어리를 주석으로 만든다.

3.5 이름

이름은 ASCII 영문자나 밑줄로 시작하고 영문자·숫자·밑줄로 이어진다. 두 가지가 다른 언어와 다르다.

첫째, **짓는 이름에는 점이 없다.** 점이 붙은 자리는 이름을 **가리키는** 경로이고 뜻은 셋뿐이다 — 모듈의 이름(`allocs.byte_allocator`), 변형의 이름(`node.lit`), 타입에 딸린 선언의 이름(`rect.area`). 값의 안을 보는 일(필드 읽기, 메서드 부르기)은 붙임 점이 아니라 `field p x` · `method s area` 같은 폼이다.

둘째, **가림(shadowing)이 없다.** 안쪽 이름이 바깥 이름과 같은 철자를 쓰면 거절된다.

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
```

실행 결과

```
$ 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
```

가려진 이름은 머리를 읽은 사람이 **다른 값을 떠올리게** 만든다. `n` 이 매개변수인지 새 지역인지를 줄마다 따져야 한다면 그것이 곧 의미의 엔트로피다. 매개변수·모듈의 이름·내장 op 의 이름·바깥 블록에 살아 있는 이름 가운데 어느 것도 가릴 수 없다. 이름을 새로 지으면 된다.

> **문. 모듈 이름과 op 이름이 같아도 되는가?**
>
> **답.** 안 된다. 모듈 `twice` 안에 op `twice` 를 두면 둘 다 `twice.twice` 와 `twice` 로 가리켜져 구별할 수 없으므로 `E-NAME-DUP` 로 거절된다. 이 책의 예제 모듈 이름이 op 이름과 조금씩 다른 이유다.

3.6 낱말은 예산이다

언어가 뜻을 정해 둔 낱말(키워드)은 목록이 닫혀 있다. 새 낱말을 하나 더하는 것은 언어가 커지는 일이므로, 기존 낱말로 표현할 수 있는 것에는 새 낱말을 주지 않는다. 대략의 갈래는 다음과 같다.

갈래	낱말
선언	`module use type newtype struct enum trait contract actor state`
op	`fn proc export unsafe extern`
지역과 흐름	`let var set return if else for while guard match case try break continue expr`
값	`make true false none be`
그 밖	`spawn send drop test expect satisfies do end`

표 3.3 — 핵심 낱말의 갈래

`add`·`len`·`neg` 같은 것은 낱말이 아니라 **내장 op** 이다. 매개변수 이름으로 쓸 수 없는 것은 같지만, 문법이 아니라 이름의 자리를 차지한다. 없앤 낱말(`in`, `loop`, `as`, `to` 따위)은 조용히 받아 주지 않고 `E-VOCAB-REMOVED` 로 무엇으로 바꾸라는지 알려 준다.

3.7 `expr` 섬의 경계

섬 안에서 쓸 수 있는 것은 표 하나가 전부다. `* /` 가 `+ -` 보다 강하고, 비교(`eq lt` 따위)가 그 아래, `and` 가 `or` 보다 강하다. 비트 연산·나머지·최솟값 따위는 섬 안에서도 전위로 적는다. 비트 연산의 우선순위는 언어마다 달라서, 섬에 넣으면 읽기 쉬워지는 것이 아니라 찾아볼 것이 늘기 때문이다.

비교는 이어 쓸 수 없다.

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
```

실행 결과

```
$ 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)`
```

수학의 `a < b < c` 는 "a 가 b 보다 작고 b 가 c 보다 작다" 이지만 많은 언어에서 그 표기는 `(a < b) < c` 로 읽힌다. 같은 글자가 사람과 기계에게 다른 뜻이면 조용히 틀리는 자리다. 진단이 제안하듯 `expr (a lt b) and (b lt c)` 로 적는다.

3.8 op 머리의 절 차례

op 의 머리는 이름 뒤에 **절**을 늘어놓은 것이다. 절마다 마침표로 닫고, 절의 차례는 하나로 정해져 있다.

차례	절	왜 이 자리인가
1	`satisfies` · `lowdoc`	이 op 이 **무엇인지**를 먼저 말한다
2	`vector` · `priority`	op 전체의 성격
3	`comptime` 입력	뒤의 입력·출력 타입이 이 이름을 쓴다
4	권한·영역 입력(`cap …` · `region …`)	누가 허락했는지가 무엇을 받는지보다 먼저 보인다
5	`using`	뒤의 데이터가 어느 할당기를 쓰는지
6	데이터 입력	op 이 받는 값
7	`output`	출력 타입은 입력의 타입 매개변수를 쓸 수 있다
8	`effects`	앞에서 받은 권한으로 **무엇을 하는지**
9	`link` · `variadic` · `asm`	바깥(C·기계어)과 잇는 자리
10	`access` · `parallel` · `reduce`	나누어 도는 방법
11	`requires` → `ensures` → `errors` → `tests`	입력 조건, 출력 약속, 실패, 시험

표 3.4 — op 머리의 절 차례(앞에서 뒤로)

모든 절을 외울 필요는 없다. 대부분의 op 은 이 가운데 너덧 개만 쓴다. 기억할 것은 원리다 — **앞의 것이 뒤의 것에 쓰이도록** 놓였다. 타입 매개변수가 입력보다 먼저, 권한이 데이터보다 먼저, 입력이 출력보다 먼저, 권한이 효과보다 먼저, 효과가 계약보다 먼저다.

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

한 줄에 이어 적어도, 절마다 줄을 나누어도 뜻은 같다(개행은 공백이다). 짧은 머리는 한 줄로, 계약이 붙는 머리는 절을 나누어 적는 것이 이 책의 버릇이다.

> **실제 사례. 출력이 맨 앞이던 때**
>
> 이 차례는 한 번에 정해지지 않았다. 한동안은 `output` 을 머리의 맨 앞에 두는 차례를 썼다 — op 이 무엇을 돌려주는지가 가장 먼저 보인다는 이유였다. 그러나 출력 타입이 입력의 타입 매개변수를 쓰는 제네릭 op 에서는 이름이 선언되기 전에 쓰이는 모양이 되고, 권한과 효과의 짝도 멀리 떨어졌다. 그래서 절끼리 서로를 쓰는 관계에 맞춰 지금의 차례로 다시 정하고 고정했다. 컴파일러는 이 차례를 순위표로 강제한다.

3.9 흔한 실수

겉모습의 실수는 대개 **다른 언어의 버릇**에서 온다. 진단이 엉뚱한 곳을 가리키는 것도 있어서, 모양을 함께 기억해 두면 좋다.

> **반례. `//` 로 주석을 단다**
>
> `examples/ch03/mistake_slash.low`

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

fn twice input a u64 . output u64 .
do
  rem ✘ `//` 는 주석 표시가 아니다 --- 이 줄과 아랫줄이 마침표 하나로 끝나는 폼 하나가
된다
  // double it
  return mul a 2 .
end
```

실행 결과

```
$ 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
```

이 언어의 주석은 `rem`(줄 주석)과 `note WHY … WHY`(여러 줄)뿐이다. `//` 는 주석이 아니어서 컴파일러는 그것을 폼의 시작으로 읽고, 폼은 **마침표가 나올 때까지** 이어진다. 그래서 `// double it` 과 다음 줄의 `return mul a 2` 가 마침표 하나로 끝나는 폼 하나가 되고, `return` 이 그 속에 삼켜진다. 진단은 "돌려주지 않는 길이 있다" 고 말하지만 원인은 주석이다. 주석을 낱말로 여는 까닭은 기호를 줄이려는 것이다 — `//` · `#` · `--` 가 언어마다 다르니, 낱말 하나로 정했다. 고치는 법: `rem double it.`

반례. 붙임 점으로 필드를 읽는다

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 ✘ 붙임 점으로 필드를 읽었다 --- 필드는 `field p x` 로 읽는다
  return p.x .
end
```

실행 결과

```
$ 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` 는 C 나 Python 에서 흔하지만 여기서는 `E-FIELD-GLUED` 다. 점은 이미 **이름을 가리키는 경로**(모듈 `allocs.bump_bytes`, 변형 `color.red`)에 쓰고 있어서, 값의 안을 보는 일까지 맡기면 `a.b` 를 볼 때마다 어느 뜻인지 따져야 한다. 필드 읽기는 `field p x` 로 적는다(10장).

반례. 머리의 절을 아무 차례로 적는다

examples/ch03/mistake_order.low

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

rem ✘ `output` 이 입력보다 먼저 왔다 --- 절의 차례는 하나다
fn twice output u64 . input a u64 .
do
  return mul a 2 .
end
```

실행 결과

```
$ 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)
```

절의 차례는 하나다(표 3.4). 차례가 자유로우면 같은 머리가 여러 모양으로 적혀, 읽는 사람이 매번 절을 찾아 헤매야 한다. 진단은 무엇이 무엇 뒤에 왔는지 말하고, `lowentc --fmt` 가 차례를 바로잡아 준다.

반례. `if` 의 몸을 `do` 없이 연다

examples/ch03/mistake_ifdo.low

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

fn pick input a u64 . output u64 .
do
  rem ✘ `if` 의 몸을 `do` 로 열지 않았다
  if gt a 3 .
    return 1 .
  end
  return 0 .
end
```

실행 결과

```
$ 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
```

Python 처럼 들여쓰기로 몸을 여는 문법이 아니다. `do` 가 없으면 `if` 폼은 조건 뒤의 마침표에서 끝나고, 아래의 `end` 는 `if` 가 아니라 **op 의 몸**을 닫아 버린다. 그러면 남은 `return 0 .` 이 선언 밖으로 떨어져 `E-TOPLEVEL`(최상위에 선언이 아닌 것이 있다)이 나오고, 몸이 일찍 닫혔으니 돌려주는 길도 어긋난다(`E-RETURN-PARTIAL`). 두 진단이 함께 보이면 빠진 `do` 를 의심한다. 고치는 법: `if gt a 3 . do`.

반례. 문자열을 작은따옴표로 쓴다

examples/ch03/mistake_quote.low

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

fn size output u64 .
do
  rem ✘ 작은따옴표는 글자 하나다 --- 문자열은 큰따옴표 "hi" 로 쓴다
  return len 'hi' .
end
```

실행 결과

```
$ 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
```

작은따옴표는 **글자 하나**의 리터럴이다(`'a'` 는 바이트 97). 두 글자를 넣으면 한 칸에 들어가지 않는다는 `E-CHAR-WIDTH` 가 나온다. 문자열은 언제나 큰따옴표로 쓴다: `len "hi"` 는 2 다. 두 따옴표의 뜻을 가른 것은 “글자 하나” 와 “바이트 여럿” 이 서로 다른 타입이기 때문이다(표 3.1).

흔한 오해. 앞자리가 0 인 수는 팔진수다

C 에서 `0755` 는 팔진수 493 이지만 Lowent 에는 팔진 표기가 없다.

examples/ch03/octal.low

```
module octal .
rem run: perm

fn perm output u64 .
do
  rem 앞자리 0 은 아무 뜻이 없다 --- 이것은 칠백오십오다
```

```
  return 0755 .
end
```

실행 결과

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

같은 글자가 언어마다 다른 수가 되는 자리라서 없앴다. 권한 비트처럼 팔진이 편한 값은 `0x1ED`(십육진)나 비트 연산으로 적는다.

3.10 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`add a (mul b c)`	전위 표기 — 이름이 먼저, 안쪽 호출은 괄호로	우선순위를 외울 필요가 없다
`… .`	떨어진 마침표가 폼(문장·절)을 닫는다	개행이 뜻을 바꾸지 않게 — 줄은 어디서 나눠도 된다
`do … end`	모든 블록	블록을 여는 모양이 하나뿐
`rem … • note WHY … WHY`	줄 주석 · 여러 줄 주석	기호 주석(`//`·`#`)이 없다 — 낱말로 연다
`42 • 0x2A • 0b101010 • 1_000`	정수 리터럴(자리가 타입을 정한다)	팔진 표기가 없다 — `0755` 는 755
`"hi\n" • 'a'`	문자열(바이트 여럿) · 글자 하나	따옴표가 곧 타입의 차이
`true • false • none`	참거짓 · 값 없음	값도 낱말로
`field p x • method s area`	필드 읽기 · 메서드 부르기	점은 이름의 경로에만 쓴다
`allocs.bump_bytes • color.red`	모듈 안의 이름 · 변형의 이름	점 하나는 “이 이름 안의 저 이름”
`expr a + b * c`	중위 섬 — 사칙·비교·`and`/`or` 만	긴 산술을 읽기 쉽게, 뜻은 전위와 같다
op 머리의 절 차례	표 3.4	앞의 것이 뒤의 것에 쓰이도록
`0b1010 • 1_000_000 • 0x1.8p1 • u'가' • U"…"`	이진 · 자리 구분 · 십육진 부동소수 · 접두사 문자·문자열	팔진은 없다 — `0755` 는 755
`text DOC … DOC • note END … END`	여러 줄 문자열(이스케이프를 풀지 않는다) · 여러 줄 주석	긴 글을 고치지 않고 그대로 싣는다

표 3.5 — 겉모습의 규칙 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

떨어진 마침표가 폼을 닫고, 개행은 공백이다. 식은 이름이 먼저 오는 전위 표기이고 중위는 `expr` 섬 안에서만 쓴다. 블록은 언제나 `do … end` 다. 이름에는 점이 없고 가림은 금지된다. 리터럴은 자리의 타입에 들어가지 않으면 거절된다. op 머리의 절은 한 차례로만 적으며, 그 차례는 앞의 것이 뒤의 것에 쓰이도록 놓였다.

제2부 — 값과 흐름

4 수 — 폭이 정해진 정수와 부동소수

이 장의 차례

4.1 정수 타입은 이름에 폭을 적는다 48
4.2 넓히기는 자동, 좁히기는 적는다 48
4.3 넘치면 멈춘다 49
4.4 나눗셈과 나머지 51
4.5 참거짓은 수가 아니다 52
4.6 비트 연산 52
4.7 부동소수 53
4.8 흔한 실수 55
4.9 이 장의 문법 한눈에 60

먼저 알아야 할 것

2장 첫 프로그램 · VM 과 네이티브가 같은 답을 내야 한다
3장 겉모습 · 리터럴은 자리의 타입에 들어가지 않으면 거절된다

돌아보기

3장에서 `let x u8 be 300 .` 은 어떻게 되었는가? C 는 같은 일을 어떻게 처리하는가?

> **답.** `E-TYPE-WIDTH` 로 거절되었다. 300 은 `u8` 의 범위 0 … 255 밖이기 때문이다. C 는 조용히 잘라 44 로 만든다. 그러면 소스에 적힌 값과 실제 값이 달라진다. 이 장은 리터럴이 아니라 **계산 결과**가 폭을 넘을 때 같은 원칙이 어떻게 적용되는지를 본다.

이 장의 필요성과 맥락

시스템 프로그램의 결함 가운데 오래되고 흔한 것이 정수의 결함이다. 넘쳐서 작은 수가 되고, 음수가 큰 양수로 바뀌고, 0 으로 나누어 미정의 동작이 된다. 대부분은 **조용히** 일어나서 한참 뒤의 엉뚱한 자리에서 드러난다. 제2부의 첫 장이 수인 것은 모든 계산이 수 위에서 일어나고, Lowent 가 "조용히" 를 없애는 방식 — 처분을 이름으로 적게 하는 방식 — 을 가장 단순한 모양으로 보여 주는 곳이 여기이기 때문이다.

이 장이 끝나면

정수 타입이 부호와 폭으로 이름 지어지고 어디서나 같은 크기라는 것을 알게 된다. 넓히기는 자동이고 좁히기는 적어야 한다는 규칙, 넘침과 0 나누기가 멈춘다는 규칙, 그리고 멈추지 않고 감거나 끝값에 서거나 값으로 알려 달라고 **이름으로** 고르는 법을 익힌다. 참거짓이 수가 아니라는 것과 비트 연산· 부동소수의 성질도 보게 된다.

이 장에서 답할 질문

1 `narrow_wrap` 이 C 와 같은 일을 한다면, 기본을 그렇게 두지 않은 이유가 무엇인가?
2 비트 연산은 왜 `expr` 섬에서 중위로 쓸 수 없는가?

4.1 정수 타입은 이름에 폭을 적는다

정수 타입의 이름은 부호와 비트 폭이다. `u` 는 부호 없음, `i` 는 부호 있음이다.

타입	폭	범위
`u8` · `u16` · `u32` · `u64`	8 · 16 · 32 · 64	$0 \cdots 2^{N}-1$
`i8` · `i16` · `i32` · `i64`	8 · 16 · 32 · 64	$-2^{N-1} \cdots 2^{N-1}-1$
`usize` · `isize`	주소 폭	실행 환경이 정한다

표 4.1 — 정수 타입과 범위

C 의 `int` 는 기계마다 크기가 다를 수 있지만, Lowent 의 `u32` 는 어디서나 32 비트다. 음수는 2 의 보수로 표현한다. `usize` 는 `u64` 와 폭이 같은 기계에서도 **다른 타입**이고 서로 자동으로 바뀌지 않는다 — 주소를 세는 수와 그냥 수를 섞는 자리를 소스에 드러내기 위해서다.

4.2 넓히기는 자동, 좁히기는 적는다

값을 잃지 않는 변환을 **넓히기**라 한다. `u8` 을 `u32` 자리에 넘기는 일은 값이 그대로이므로 컴파일러가 알아서 한다. 부호가 섞여도 값을 지키는 넓히기가 있으면 계산할 수 있다 — `u8` 은 `i16` 안에 모두 들어간다.

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
```

실행 결과

```
$ 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` 는 `u8` 과 `i16` 을 더했고 결과는 넓은 쪽인 `i16` 이다. 넓히기를 눈에 보이게 적고 싶으면 `widen u64 x` 로 적는다. 같은 폭의 `i32` 와 `u32` 는 어느 쪽도 다른 쪽에 다 들어가지 않으므로 섞을 수 없다.

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
```

실행 결과

```
$ 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)
```

값을 잃을 수 있는 변환은 **좁히기**이고 반드시 적는다. 같은 파일의 `shrink` 는 `narrow u8 x` 로 `u64` 를 `u8` 로 줄인다. 200 은 들어가서 그대로 나오지만, 300 은 들어가지 않으므로 실행이 멈춘다 (`E-VM-CAST`). 멈추지 않기를 원하면 원하는 처분을 이름으로 고른다. `narrow_sat` 는 끝값 255 에 서고, `narrow_wrap` 은 C 처럼 감아서 44 를 준다. `narrow_try` 는 들어가지 않으면 `none` 을 준다.

> **문.** `narrow_wrap` **이 C 와 같은 일을 한다면, 기본을 그렇게 두지 않은 이유가 무엇인가?**
>
> **답.** 감기가 **의도였는지 실수였는지**를 소스에 남기려는 것이다. 기본이 감기면 `narrow u8 x` 가 44 를 준 자리를 읽는 사람이 그것이 원하던 값인지 알 수 없다. 기본이 멈춤이면, 감기를 원한 저자는 `narrow_wrap` 이라고 **적었을** 것이다. 적힌 이름이 곧 저자의 의도에 대한 기록이다.

4.3 넘치면 멈춘다

산술은 **선언된 폭에서** 일어난다. 결과가 폭에 들어가지 않으면 멈춘다(트랩).

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
```

실행 결과

```
$ 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` 는 255 로 들어가지만 `bump 255` 는 256 이 되어야 하므로 VM 이 `E-VM-OVERFLOW` 로 멈춘다. 네이티브도 같은 자리에서 멈추고 비영 종료 코드를 낸다(검증 스크립트는 둘 다 멈추는지를 확인한다). 처분을 바꾸려면 넓히기와 마찬가지로 이름을 고른다.

처분	op	넘치면
멈춤(기본)	`add` · `sub` · `mul`	실행이 멈춘다
감김	`wrap_add` · `wrap_sub` · `wrap_mul`	폭 안에서 돌아 감는다
포화	`sat_add` · `sat_sub` · `sat_mul`	그 폭의 끝값에 선다
값으로	`chk_add` · `chk_sub` · `chk_mul`	`option` 으로 알린다

표 4.2 — 넘침의 처분을 이름으로 고른다

"값으로" 알리는 쪽은 넘침을 프로그램이 다룰 수 있는 사건으로 만든다.

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
```

실행 결과

```
$ 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` 은 `option u8` 을 준다. 넘치면 `none` 이고, `guard is_some r . else …` 가 그 경우를 먼저 처리한다. 이 모양은 11장에서 다시 만난다.

> **흔한 오해. 넘침을 검사하면 느려진다**
>
> 모든 덧셈에 검사가 남는다면 그렇다. 그러나 계약이나 범위 타입으로 값의 범위가 증명되는 자리에서는 검사가 번역할 때 지워진다(14장). 반복 변수처럼 범위가 뻔한 자리도 컴파일러의 구간 분석이 걷어낸다. 남는 검사는 **정말로 넘칠 수 있는** 자리의 검사이고, 그 자리의 검사는 비용이 아니라 결함을 막는 값이다.

4.4 나눗셈과 나머지

정수를 0 으로 나누면 멈춘다. 부호 있는 나눗셈은 0 쪽으로 버리고, 나머지 `mod` 의 부호는 **나누는 수**를 따른다.

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
```

실행 결과

```
$ 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` 는 −3 이고 `modulo -7 2` 는 1 이다. 나머지가 나누는 수의 부호를 따르므로, 양수 `n` 으로 나눈 `mod h n` 은 언제나 0 이상 `n` 미만이다. 해시 테이블이 슬롯을 고를 때 `index slots (mod h n)` 의 경계 검사가 지워지는 근거가 이 성질이고, 이 성질은 Coq 로 증명되어 있다(40장).

부호 있는 나눗셈이 넘치는 경우는 `MIN / -1` 하나뿐이고, 이것도 멈춘다. C 에서는 미정의 동작이다.

0 이 아님을 한 번 확인하고 그 사실을 들고 다니는 길도 있다. `nonzero_of b` 는 `option nonzero u32` 를 주고, 그 안의 값은 `div_nz` 에 넘길 수 있다. `div_nz` 자리에는 0 검사가 남지 않는다. 검사를 없앤 것이 아니라 **한 자리로 옮기고 그 결과를 타입에 실은** 것이다.

4.5 참거짓은 수가 아니다

`bool` 은 `true` 와 `false` 두 값이다. 정수를 조건 자리에 쓰면 거절된다.

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
```

실행 결과

```
$ 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 의 `if (n)` 은 "n 이 0 이 아니다" 로 읽히지만, 읽는 사람은 그것이 "n 이 있다" 인지 "n 이 참이다" 인지 문맥으로 짐작해야 한다. Lowent 는 묻는 것을 적게 한다 — `if gt n 0 .` 이다. 반대로 `bool` 을 수처럼 더할 수도 없다. `and`·`or`·`not` 은 참거짓만 받고, `and` 와 `or` 는 앞쪽만으로 답이 정해지면 뒤쪽을 계산하지 않는다.

4.6 비트 연산

비트 연산은 값을 비트의 나열로 보고 다룬다. 기호 대신 낱말이다.

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
```

실행 결과

```
$ 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` — 비트별 논리. `flip 12` 가 243 인 것은 `u8` 이기 때문이다. 비트 연산의 답은 언제나 타입의 폭에 매여 있다.
- `shl`·`shr` 은 옮기기, `rotl`·`rotr` 은 돌리기다. `spin 129 1` 이 3 인 것은 맨 앞의 1 이 맨 뒤로 돌아 들어왔기 때문이고, `shl` 이었다면 그 1 은 버려진다.
- `shr` 은 부호에 따라 뜻이 다르다. `-8` 과 `4294967288` 은 32 비트에서 같은 비트열이지만, 부호 있는 쪽은 부호 비트가 들어와 −4 가 되고 부호 없는 쪽은 0 이 들어와 2147483644 가 된다.
- `count_ones`·`leading_zeros`·`trailing_zeros` 는 비트를 세고, `byte_swap` 은 바이트 차례를 뒤집는다.
- `clmul_lo`·`clmul_hi` 는 **캐리 없는 곱셈**이다 — 자리 올림 대신 배타합으로 더하는 곱셈이고, 암호와 검사합(GF(2^128)·CRC)이 이것 위에 선다. 답이 128 비트라 낱말 둘로 나온다. 기계에 그 명령이 있으면 처리기가 그것으로 내리고(`lowentc --hw pclmul` 또는 `auto`), 없으면 같은 답을 내는 셈으로 내린다 — **뜻은 하나**이고 빠르기만 다르다.

옮기는 칸 수가 타입의 폭 이상이면 멈춘다. C 에서는 미정의 동작인 자리다. 칸 수를 폭 안으로 감싸기를 원하면 `wrap_shl`·`wrap_shr` 을 쓴다.

> **문. 비트 연산은 왜 `expr` 셈에서 중위로 쓸 수 없는가?**
>
> **답.** C 에서 `a & b == c` 는 `a & (b == c)` 로 묶인다. 오래 쓴 사람도 걸리는 함정이고, 언어마다 비트 연산의 우선순위가 다르다. 셈에 넣으면 읽기 쉬워지는 대신 찾아볼 것이 는다. 전위로 적으면 괄호가 답을 말하므로 틀릴 수 없다.

4.7 부동소수

`f32` 와 `f64` 는 IEEE 754 의 이진 32·64 비트 형식이다. 연산은 피연산자의 폭에서 일어나고, 정수와 부동소수는 서로 자동으로 바뀌지 않는다.

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
```

실행 결과

```
$ 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
```

VM 은 부동소수 결과를 짧게 보여 준다(`0.333333`). 부동소수를 0 으로 나누면 멈추지 않고 IEEE 754 가 정한 대로 무한이 된다. `sqrt`·`sin`·`exp` 같은 초월 함수는 부동소수 전용이고 운영체제가 있는 기계에서만 쓸 수 있다. 이 함수들의 마지막 비트가 어떤 값인지는 기계와 셈 라이브러리가 정한다.

흔한 오해. 부동소수도 `eq` 로 비교하면 된다

> 0.1 과 0.2 를 더한 값은 0.3 과 비트가 다르다. 부동소수의 같음은 대개 “충분히 가깝다” 로 물어야 하고, 표준 라이브러리의 `math` 모듈이 그 물음(`close`)을 준다(32장). 부동소수의 반올림· NaN·−0 같은 성질은 이 언어의 증명이 다루지 않은 자리이기도 하다(50장).

`f32` 와 크기 타입도 같은 규칙을 따른다.

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 · isize — 기계의 주소 폭(64 비트 기계에서 64 비트). 길이와 색인을 C 와 주고받
을 때 쓴다
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 — 32 비트 부동소수. 메모리가 절반이고 정밀도는 약 일곱 자리
fn third32 input x f32 . output f32 . do
  return div x 3.0 .
end
```

```
rem 같은 1/3 도 f32 와 f64 는 다른 수다 — 넓혀서 비교하면 거짓
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
```

실행 결과

```
$ 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`·`isize` 는 기계의 주소 폭을 가진 부호 없는·있는 정수다. C 의 `size_t`·`ptrdiff_t` 와 길이·색인을 주고받을 때 쓴다(29장). 폭이 기계마다 달라질 수 있으므로, 폭이 정해진 계산에는 `u64`·`i64` 를 쓴다. `cast isize n` 처럼 부호가 다른 타입으로는 이름 붙은 연산으로 옮긴다.
- `f32` 는 메모리가 절반이고 유효 숫자가 약 일곱 자리다. 큰 배열이나 그래픽처럼 정밀도보다 양이 중요한 곳에 쓴다.
- `same_third` 는 같은 1/3 을 `f32` 와 `f64` 로 셈해 넓힌 뒤 비교한다. 거짓(0)이다. VM 이 둘 다 `0.333333` 으로 짧게 보여 주어도 둘은 다른 수다.

4.8 흔한 실수

수의 실수는 번역을 통과하고 **실행 중에** 드러나는 것이 많다. 다른 언어라면 조용히 틀린 값이 나올 자리에서 Lowent 는 멈춘다. 멈춘 자리가 곧 고칠 자리다.

> **반례. 둘을 더한 뒤 반으로 나눠 가운데 값을 구한다**

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 ✘ 둘을 먼저 더하면 300 이 되어 u8 에 들어가지 않는다
  return div (add a b) 2 .
end
```

실행 결과

```
$ 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` 은 300 이고 `u8` 은 255 까지다. 나누기 전에 이미 넘친다. 이진 탐색의 `(lo + hi) / 2` 가 수십 년 동안 여러 라이브러리에서 터진 바로 그 결함이다. C 라면 감긴 값 44 를 반으로 나눈 22 가 조용히 나왔을 것이다. 고치는 법은 계산을 **넓은 폭에서** 하는 것이다.

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 한 폭 넓혀서 더한다 --- u16 은 510 까지 담는다
  let both u16 be add (widen u16 a) (widen u16 b) .
  rem 둘로 나눈 값은 다시 u8 에 들어간다는 것을 우리는 알고, narrow 가 그것을 확인한다
  return narrow u8 (div both 2) .
end
```

실행 결과

```
$ 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 u16` 으로 올려서 더하고, 나눈 뒤 `narrow u8` 로 내린다. 가운데 값은 언제나 `u8` 에 들어가므로 좁히기가 멈출 일은 없고, 혹시 계산을 잘못 바꾸면 그 자리에서 멈춰 알려 준다. (`lo <= hi` 가 보장되면 `add lo (div (sub hi lo) 2)` 도 넘치지 않는다.)

반례. 부호 없는 수에서 큰 수를 뺀다

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 ✘ 부호 없는 수는 0 아래로 내려가지 않는다 --- 3 - 5 는 음수가 아니라 멈춤이다
  return sub a b .
end
```

실행 결과

```
$ 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` 는 0 아래가 없다. `3 - 5` 는 −2 가 아니라 **넘침**이고, 그래서 멈춘다(C 라면 18446744073709551614 가 나온다). 차이의 크기가 알고 싶으면 큰 쪽에서 작은 쪽을 뺀다 — `if ge a b . do return sub a b . end return sub b a .` — 음수가 뜻이 있는 값이면 처음부터 `i64` 로 계산한다.

반례. 반복 변수를 너무 좁은 타입으로 둔다

examples/ch04/mistake_narrowloop.low

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

fn tally output u64 .
do
  rem ✘ u8 반복 변수는 255 다음이 없다 --- 256 번째 `add i 1` 에서 멈춘다
  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
```

실행 결과

```
$ 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)
```

`u8` 은 256 에 닿을 수 없으니 `lt i 256` 은 영원히 참이다. C 라면 `i` 가 255 에서 0 으로 감겨 **끝나지 않는 반복**이 되었을 것이다. Lowent 는 그 비교를 번역 때 `E-TYPE-WIDTH` 로 거절한다 — 자리에 안 들어가는 값과의 비교는 입력과 무관하게 답이 같으므로 아무것도 가르지 않는다. 반복 변수는 셀 수 있는 가장 큰 값보다 넓은 타입(`u64`)으로 둔다.

반례. 부동소수를 정수 리터럴로 계산한다

examples/ch04/mistake_floatint.low

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

fn half input x f64 . output f64 .
do
  rem ✘ f64 를 정수 리터럴 2 로 나눴다 --- 부동소수 자리에는 2.0 을 쓴다
  return div x 2 .
end
```

실행 결과

```
$ 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`)
```

정수와 부동소수는 서로 자동으로 바뀌지 않는다. `f64` 를 나누려면 리터럴도 부동소수로 `2.0` 이라고 적는다. 섞어 적으면 `E-TYPE-MIX` 로 번역 때 거절한다 — 전에는 **실행 중에** `E-VM-TYPE` 으로 멈췄다.

반례. `--run` 인자로 `u64` 의 가장 큰 값을 넘겨 시험한다

examples/ch04/mistake_runmax.low

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

rem ✘ u64 의 위쪽 절반을 `--run` 인자로 주면 경계에서 거절된다 — 구간 분석이 u64 의 상한
을 i64 최댓값으로 믿기 때문이다
fn echo input n u64 . output u64 .
do
  return n .
end

rem 경계값은 시험 안에서 리터럴로 적는다
test largest_u64
do
  let m u64 be 18446744073709551615 .
  expect gt m 9223372036854775807 .
  expect eq (div m 2) 9223372036854775807 .
end
```

실행 결과

```
$ 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` 는 경계에서 거절된다(E-VM-CONTRACT). 2026-09-16 까지는 조용히 9223372036854775807 로 잘려 들어갔다 — 다른 수를 준 것이고, 두 뒤끝이 똑같이 그랬으므로 대조로도 드러나지 않았다. 지금은 자르지 않고 거절한다. 까닭은 구간 분석이 `u64` 의 상한을 `i64` 의 최댓값으로 적어 두기 때문이다 — 그 위의 값이 들어오면 분석이 스스로 무너진다. 프로그램 안의 리터럴은 온전하므로 경계값 시험은 `largest_u64` 처럼 거기서 적고, 결과 표시는 이제 부호 없이 찍힌다.

흔한 오해. `f32` 와 `f64` 는 같은 수를 담는다

examples/ch04/mistake_f32literal.low

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

rem ✘ 리터럴을 그대로 담은 f32 — 이 판에서는 32 비트로 반올림되지 않아 f64 의 0.1 과 같
다고 나온다
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 셈을 한 번 거친 f32(0.1 + 0)는 32 비트로 반올림되어 다르다고 나온다
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
```

실행 결과

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

`stored` 와 `computed` 가 둘 다 거짓이다. `f32` 자리에 담긴 0.1 은 32 비트로 반올림되므로, 넓혀도 `f64` 의 0.1 과 다른 수다. 2026-09-16 까지는 리터럴(과 `--run` 으로 건넨 인자)이 반올림되지 않아 `stored` 가 참이었다 — 같은 타입의 값이 어디서 왔느냐에 따라 다른 정밀도를 갖던 자리다. 정밀도가 중요한 코드는 폭을 적어서 고르고, 두 폭을 견줄 때는 어느 쪽으로 옮길지 `widen`·`narrow` 로 적는다.

흔한 오해. `div 7 2` 는 3.5 다

정수 나눗셈은 몫만 남기고 소수점 아래를 버린다.

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 정수 나눗셈은 소수점 아래를 버린다 --- 7 / 2 는 3
  return div (add a b) 2 .
end

fn avg_f input a f64 . input b f64 . output f64 .
do
  rem 소수점 아래가 필요하면 처음부터 부동소수로 계산한다
  return div (add a b) 2.0 .
end
```

실행 결과

```
$ 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` 는 3 이다. 소수점 아래가 필요하면 입력부터 부동소수로 받아 `2.0` 으로 나눈다. 반올림이 필요하면 `add n (div d 2)` 를 더한 뒤 나누는 식으로 **적어서** 고른다 — 언어가 대신 반올림하지 않는다.

부동소수의 나머지는 정수의 `mod` 와 이름을 갈라 `fmod` 로 적는다.

examples/ch04/floatmod.low

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

rem 부동소수의 나머지는 `fmod` --- 정수의 `mod` 와 이름을 가른다
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
```

실행 결과

```
$ 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` 은 1.5, `mod 7 2` 는 1 이다. 한 이름에 두 갈래를 싣지 않는 것은 `div` 와 같은 까닭이다 — 무엇을 셈하는지가 이름에서 보인다.

4.9 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`u8` … `u64` · `i8` … `i64` · `usize` · `isize`	폭이 이름에 적힌 정수	기계마다 크기가 달라지지 않게
`f32` · `f64`	IEEE 754 부동소수	정수와 섞이지 않는다 — 리터럴도 `2.0`
`widen u64 x`	넓히기(값을 잃지 않음)	자동으로도 일어나지만 보이게 적을 수 있다
`narrow u8 x`	좁히기 — 안 들어가면 멈춘다	값이 조용히 바뀌지 않게
`narrow_sat` · `narrow_wrap` · `narrow_try`	끝값에 서기 · 감기 · `option` 으로	처분을 이름으로 고른다
`add` · `sub` · `mul`	넘치면 멈추는 산술	넘침은 기본이 결함
`wrap_add` · `sat_add` · `chk_add`	감김 · 포화 · `option`	의도한 처분이 소스에 남는다
`div` · `mod`	몫(0 쪽으로 버림) · 나머지(나누는 수의 부호)	0 나누기와 `MIN / -1` 은 멈춘다
`nonzero_of b` · `div_nz`	0 이 아님을 한 번 확인해 타입에 싣기	검사를 한 자리로 옮긴다
`bit_and` · `bit_or` · `bit_xor` · `bit_not`	비트별 논리	기호 우선순위의 함정이 없다
`shl` · `shr` · `rotl` · `rotr`	옮기기 · 돌리기	폭 이상 옮기면 멈춘다
`clmul_lo` · `clmul_hi`	캐리 없는 곱셈 — 답은 낱말 둘	기계 명령이 있으면 그것으로, 없으면 셈으로

`count_ones` · `leading_zeros` · `trailing_zeros` · `byte_swap`	비트 세기 · 바이트 뒤집기	흔한 일은 이름 하나로
`true` · `false` · `and` · `or` · `not`	참거짓과 그 논리	수를 조건에 쓰지 않는다
`usize` · `isize` · `f32`	주소 폭 정수 · 32 비트 부동소수	C 와 길이를 주고받을 때 · 정밀도보다 양이 중요할 때

표 4.3 — 수의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

정수 타입은 부호와 폭을 이름에 적고 어디서나 같은 크기다. 넓히기는 자동이고 좁히기는 `narrow` 로 적는다. 넘침·0 나누기·범위 밖 좁히기·폭 이상의 옮기기는 멈추며, 다른 처분(감김·포화·값으로)은 `wrap_`·`sat_`·`chk_`·`narrow_try` 같은 이름으로 고른다. `mod` 의 부호는 나누는 수를 따른다. 참거짓은 수가 아니고, 비트 연산은 낱말로 적으며 폭과 부호가 뜻을 정한다.

5 op — fn 과 proc

이 장의 차례

5.1 두 갈래 62
5.2 순수함은 관측으로 정한다 63
5.3 fn 에는 effects 를 적지 않는다 64
5.4 효과는 부르는 쪽으로 번진다 65
5.5 매개변수와 반환 66
5.6 neg 만 단항이다 66
5.7 수식어 67
5.8 흔한 실수 67
5.9 이 장의 문법 한눈에 70

먼저 알아야 할 것

3장 겉모습 · op 머리의 절 차례
4장 수 · 넘침은 멈추고 처분은 이름으로 고른다

돌아보기

3장의 절 차례표에서 output 과 effects 는 각각 무엇의 뒤에 오는가? 왜 그 자리인가?

> **답.** output 은 데이터 입력 뒤에 온다. 출력 타입이 입력의 타입 매개변수를 쓸 수 있기 때문이다. effects 는 output 뒤, 계약 앞에 온다. 앞에서 받은 권한으로 무엇을 하는지를 적는 자리이기 때문이다. 이 장은 그 머리를 가진 실행 단위 — op — 가 두 갈래로 나뉘는 까닭을 다룬다.

이 장의 필요성과 맥락

Lowent 프로그램의 기본 단위는 op 이고, op 은 언제나 fn 이거나 proc 이다. 이 갈래는 제4부의 효과 체계, 제6부의 트레이트, 제7부의 병렬이 모두 기대는 바닥이다. 순수한 op 은 결과를 기억해 두어도, 차례를 바꾸어도, 나누어 돌려도 되기 때문이다. 그래서 수를 다룬 바로 다음, 지역과 흐름보다 먼저 op 의 두 갈래를 세운다.

이 장이 끝나면

fn 과 proc 의 차이, 그리고 순수함이 **관측적**이라는 뜻(안에 갇힌 변이는 순수)을 알게 된다. 매개변수를 input 절로 받고 output 으로 돌려주는 법, 재귀, neg 가 유일한 단항 산술이라는 것을 익힌다. 순수한 op 이 비순수한 op 을 부를 때와 mut 매개변수로 호출자에게 쓸 때 무엇이 거절되는지도 보게 된다.

이 장에서 답할 질문

1 proc 인데 실제로 아무 효과도 내지 않으면 어떻게 되는가?

5.1 두 갈래

op(operation)은 이름과 계약을 가진 실행 단위다. 표면에서는 언제나 두 낱말 가운데 하나로 연다.

- `fn` — **순수한** op. 같은 입력이면 같은 결과를 내고 바깥에 흔적을 남기지 않는다.
- `proc` — 효과를 낼 수 있는 op. 무슨 효과를 내는지 `effects` 절에 적는다.

기본값은 없다. `fn` 인지 `proc` 인지는 이름 앞에 늘 적혀 있다. `add` 나 `len` 같은 내장 연산도 op 이다 — 언어가 미리 만들어 둔 op 이라서 **내장 op** 이라 부른다.

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 지역 가변 상태는 op 안에 갇혀 있으므로 fn 도 쓸 수 있다
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
```

실행 결과

```
$ 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
```

세 op 이 모두 `fn` 이다. `add3` 은 인자 셋을 받아 더하고, `fact` 는 자기 자신을 부르는 재귀다. `fact` 의 `requires le n 20 .` 은 21! 이 `u64` 를 넘기 때문에 붙인 계약이다. 이 계약이 없으면 `fact 21` 은 곱셈에서 멈춘다(4장). 계약이 있으면 **진입에서** 멈추고, 멈춘 자리가 부르는 쪽의 잘못임을 알린다.

5.2 순수함은 관측으로 정한다

`running_total` 을 다시 보자. 본문에 `var total` 이 있고 `set` 으로 값을 바꾼다. 그런데도 `fn` 이다. Lowent 의 순수함은 **관측적**이기 때문이다. op 안에 갇힌 지역 변수의 변이는 바깥에서 볼 수 없으므로 순수로 친다. 기계어 수준에서 메모리에 쓰는 일이 일어나도, 호출자가 관측할 수 없으면 효과가 아니다.

반대로 호출자가 볼 수 있는 쓰기는 효과다. `mut` 매개변수를 통해 쓰는 `fn` 은 거절된다.

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
```

실행 결과

```
$ 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)
```

진단이 설명하는 대로, `fn` 은 강제되는 순수 계약이다. 호출자는 `fn` 의 결과를 기억해 두거나(memoise), 부르는 차례를 바꾸거나, 결과를 쓰지 않으면 호출을 지워도 된다. 호출자의 저장소를 바꾸는 op 에는 그 어느 것도 할 수 없다. 그래서 `proc` 이어야 한다.

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
```

실행 결과

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

`effects state .` 는 이 op 이 자기 바깥의 상태를 고친다는 선언이다. VM 이 보여 주는 `arg0 (written) = [0,8,9]` 가 그 흔적이다 — 호출자가 넘긴 슬라이스의 첫 원소가 바뀌었다.

> **흔한 오해. `proc` 은 `fn` 보다 느리다**
>
> 갈래는 속도와 상관이 없다. 둘 다 같은 방식으로 C 로 내려간다. 차이는 **컴파일러와 호출자가 할 수 있는 일**에 있다. `fn` 에는 차례 바꾸기·공통 부분식 제거·기억해 두기 같은 최적화가 허락되고, `proc` 에는 허락되지 않는다. 순수한 일을 `proc` 으로 적으면 그 기회를 스스로 버리는 것이다.

5.3 `fn` 에는 `effects` 를 적지 않는다

`fn` 의 효과는 이미 `none` 이다. 그래서 `effects none .` 을 덧붙이면 거절된다.

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
```

실행 결과

```
$ 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.)
```

뜻이 틀려서가 아니라 **한 뜻에 두 표기**가 되기 때문이다. `proc` 에서는 사정이 다르다. `proc` 의 `effects` 절은 **좁히는** 일을 한다. 절이 없는 `proc` 은 좁히지 않은 것으로 읽힌다 — 입출력·할당·상태를 모두 할 수 있다고 본다. 그러니 `proc` 에는 절을 적는 편이 낫고, 적은 것보다 많은 일을 하면 거절되며, 적어 놓고 하지 않는 효과도 알려진다.

> **문. `proc` 인데 실제로 아무 효과도 내지 않으면 어떻게 되는가?**
>
> **답.** `proc` 에 `effects none .` 을 적는 것은 허락된다. 지역 가변 상태나 반복을 강조하고 싶어서 `proc` 으로 두는 경우가 옛 코드에 있다. 그러나 관측되는 효과가 없다면 `fn` 으로 적는 것이 이 언어의 권하는 모양이다. 순수함이 머리에 드러나야 호출자와 컴파일러가 그것을 쓸 수 있다.

5.4 효과는 부르는 쪽으로 번진다

op 을 부르면 그 op 의 효과가 부르는 쪽의 효과에 더해진다. 순수한 `fn` 이 `io` 를 내는 `proc` 을 부르면, 그 `fn` 은 더 이상 순수하지 않다.

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
```

실행 결과

```
$ 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` 는 스스로 출력하지 않는다. 출력하는 것은 `say` 다. 그래도 거절된다. 이 규칙 덕분에 op 의 머리만 보고도 그 op 이 부르는 모든 op 의 효과를 알 수 있다. 효과를 숨긴 채 전달하는 길이 없다. 어떤 효과가 어떤 권한과 짝지어지는지는 15·16장에서 다룬다.

5.5 매개변수와 반환

op 하나를 조각조각 뜯어 보면 이렇다. 처음 보는 사람이 가장 헷갈리는 것은 **절마다 마침표로 끝난다** 는 점이다. 마침표는 문장의 끝처럼 "이 절은 여기까지" 를 알린다.

```
fn   add3   input a i32 .   input b i32 .   input c i32 .   output i32 .
|    |      └ 매개변수 a 의 타입은 i32 ┘                  └ 돌려주는 값의 타입
|    └ 이름 (부를 때 쓴다)
└ 갈래: fn(순수) 또는 proc(효과를 낼 수 있음)
do                          ← 몸이 여기서 시작한다
  return add (add a b) c .  ← 계산해서 돌려준다. 문장도 마침표로 끝난다
end                         ← 몸이 끝나고, 선언 전체도 끝난다(뒤에 점을 찍지 않는다)
```

부를 때는 이름 뒤에 인자를 **공백으로** 늘어놓는다. `add3 1 2 3` 이 한 번의 호출이고, 괄호는 호출 하나를 다른 호출의 인자로 넣을 때만 쓴다 — `add3 (add3 1 2 3) 4 5`. 괄호가 곧 "이 안이 값 하나" 라는 표시다. C 처럼 `add3(1, 2, 3)` 으로 쓰는 법은 없다. 쉼표는 이 언어에서 뜻이 없는 기호이고, 그래서 거절된다(아래 «흔한 실수»).

매개변수는 `input <이름> <타입> .` 한 절에 하나다. 여럿이면 절을 되풀이한다. 부르는 쪽은 절의 차례대로 인자를 준다. 돌려주는 값은 `output <타입> .` 하나이고, 본문에서 `return <식> .` 으로 돌려준다.

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

타입 앞에 붙는 낱말이 소유와 가변과 유무를 말한다. 자주 만나는 것은 다음과 같다.

적는 것	뜻
`input xs slice u8 .`	남의 바이트를 읽기로 빌린다
`input xs mut slice u8 .`	남의 바이트에 쓸 수 있게 빌린다(`proc` 이어야 한다)
`input p ref point .` · `mut_ref point`	값 하나를 읽기로·쓰기로 빌린다(12장)
`input o option u64 .`	있을 수도 없을 수도 있는 값(11장)
`input out cap io .`	권한(16장)
`input comptime t type .`	번역할 때 정해지는 타입(22장)

표 5.1 — 매개변수 타입에 붙는 낱말

반환값은 하나다. 여럿을 돌려주려면 `struct` 로 묶거나(10장), 호출자가 넘긴 `mut` 슬라이스에 쓴다.

5.6 `neg` 만 단항이다

사칙 연산 `add`·`sub`·`mul`·`div`·`mod` 는 인자를 둘 받는다. 산술에서 인자를 하나 받는 것은 부호를 뒤집는 `neg` 뿐이다.

`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
```

실행 결과

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

`neg` 는 부호 있는 타입의 일이다. 부호 없는 값에 쓰면 넘침이다. `abs` 는 절댓값이고, `i64` 의 가장 작은 값처럼 양수로 표현할 수 없는 값을 주면 멈춘다. `expr` 섬 안에도 단항 연산자는 없어서, 부호를 뒤집으려면 `expr 0 - a` 로 적거나 `(neg a)` 를 괄호로 부른다.

5.7 수식어

op 머리 앞에 수식어를 붙일 수 있다.

- `export` — 다른 모듈에서 부를 수 있게 내보낸다. 네이티브로 낼 때는 C 에서 부를 수 있는 심볼이 된다. 붙이지 않은 op 은 그 모듈 안에서만 쓴다. 내보내는 것은 선택이지 부작용이 아니어야 하기 때문이다.
- `extern` — 구현이 바깥(C)에 있다. 본문 대신 `link` 절로 이름을 댄다(29장).
- `unsafe` — 언어가 검사할 수 없는 일을 한다. `effects unsafe` 와 짝이다.

```
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 흔한 실수

처음 op 을 쓸 때 누구나 한 번쯤 밟는 자리들이다. 모두 번역할 때 걸리므로, 진단 코드를 보고 이 절로 돌아오면 된다.

> **반례. C 처럼 괄호와 쉼표로 부른다**
>
> 다른 언어에 익숙하면 손이 먼저 `add3(1, 2, 3)` 을 친다.
>
> `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 ✘ C 처럼 괄호와 쉼표로 불렀다 --- Lowent 의 호출은 `이름 인자 인자 …` 다
  return add3(1, 2, 3) .
end
```

실행 결과

```
$ 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.
```

Lowent 의 호출은 **이름 다음에 인자를 공백으로 늘어놓는** 전위 표기다. 쉼표는 한때 쓰였다가 없앤 기호라서 `E-VOCAB-REMOVED` 가 나온다. 괄호·쉼표 없이 부르는 까닭은 기호를 줄여 누구나(휴대전화 자판으로도) 같은 모양으로 쓰게 하려는 것이다. 고치는 법: `return add3 1 2 3 .`

반례. 인자 수가 `input` 절 수와 다르다

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 ✘ input 절이 셋인데 인자를 둘만 줬다
  return add3 1 2 .
end
```

실행 결과

```
$ 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
```

`input` 절 하나가 인자 하나다. 기본값 인자나 인자 수가 달라지는 op 은 없다 — 머리를 보면 부르는 모양이 하나로 정해져야 읽는 사람이 헷갈리지 않기 때문이다. 진단이 몇 개를 받고 몇 개를 주었는지 말해 준다.

반례. 어느 갈래에서 `return` 을 빠뜨린다

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 ✘ a 가 0 이하일 때 돌려줄 값이 없다 --- 끝까지 내려오는 길이 있다
end
```

실행 결과

```
$ 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
```

`output` 을 적은 op 은 **모든 길**에서 값을 돌려주어야 한다. 옛 도구는 빠진 자리에서 조용히 0 을 돌려주었는데, 그 0 은 소스 어디에도 적혀 있지 않은 값이라 버그를 숨겼다. 그래서 지금은 `E-RETURN-PARTIAL` 로 거절한다. 고치는 법은 `if` 뒤에 나머지 경우의 `return` 을 적는 것이다 — 예를 들면 `return 0 .` 을 `end` 앞에 둔다.

반례. 내장 op 의 이름으로 op 을 짓는다

examples/ch05/mistake_builtin.low

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

rem ✘ `min` 은 이미 내장 op 의 이름이다 --- 이 선언은 영영 불리지 않는다
fn min input a u64 . input b u64 . output u64 .
do
  return add a b .
end
```

실행 결과

```
$ 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
```

이름 공간은 평평해서 같은 이름이 둘일 수 없다. 내장 op(`add`·`min`·`len`·`ok` …)의 이름을 쓰면 그 선언은 영영 불리지 않으므로 `E-NAME-BUILTIN` 으로 알려 준다. 지역 이름에도 같은 규칙이 붙는다. 내장 이름의 목록은 부록 A 에 있다.

흔한 오해. 결과를 받지 않아도 `fn` 을 부르면 무언가 일어난다

`fn` 은 순수하므로 결과를 버리면 **아무 일도 일어나지 않은 것**과 같다. 컴파일러는 그런 호출을 지워도 된다.

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 결과를 받지 않은 fn 호출 --- 순수하므로 아무 흔적도 남기지 않는다
  twice 5 .
  rem 결과를 쓰려면 이름에 묶는다
  let t u64 be twice 5 .
  return t .
end
```

실행 결과

```
$ lowentc --run caller discard.low
caller() = 10
```

첫 `twice 5 .` 는 계산만 하고 흔적을 남기지 않는다. 답이 `10` 인 것은 둘째 줄에서 결과를 `t` 에 묶었기 때문이다. 무언가를 **하려는** 호출(출력·쓰기)이라면 그것은 `proc` 이어야 하고, 그러면 머리에 효과가 적힌다.

5.9 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`fn f input a T . output R . do … end`	순수한 op	결과를 기억·재배치·생략해도 되는지 머리만 보고 알게
`proc f … effects E . do … end`	효과를 낼 수 있는 op	무엇을 하는지(`E`)가 머리에 드러나게
`input x T .`	매개변수 하나	절마다 하나라 이름과 타입이 줄마다 보인다
`output T .` · `output void .`	돌려주는 값의 타입 · 돌려주는 값 없음	반환은 하나 — 여럿이면 `struct` 로 묶는다

`return e .`	값을 돌려주고 끝낸다	모든 길에서 적어야 한다(E-RETURN-PARTIAL)
`f a b c`	부르기 — 이름 뒤에 인자를 공백으로	괄호·쉼표 없는 전위 표기
`f (g a) b`	호출을 인자로 넣기	괄호는 “이 안이 값 하나” 라는 표시
`requires c .`	부르는 쪽이 지켜야 할 조건	어긴 쪽의 잘못임을 진입에서 알린다(14장)
`effects state .`	호출자의 저장소를 고친다	`mut` 매개변수로 쓰는 op 의 머리
`neg a`	부호 뒤집기(유일한 단항 산술)	빼기와 뒤섞이지 않게 이름으로
`export` · `extern` · `unsafe`	내보내기 · 몸이 C 에 · 검사 못 하는 일	할 수 있는 일이 머리 앞에 보이게

표 5.2 — op 의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

op 은 `fn` 이거나 `proc` 이고 갈래는 언제나 적는다. 순수함은 관측적이어서 op 안에 갇힌 변이는 순수로 치지만, `mut` 매개변수로 호출자에게 보이는 쓰기는 `proc` 이어야 한다. `fn` 에는 `effects` 를 적지 않고, `proc` 의 `effects` 는 좁히는 절이다. 효과는 부르는 쪽으로 번진다. 매개변수는 절마다 하나, 반환은 하나이며, 산술의 단항은 `neg` 뿐이다.

6 지역 — `let` 과 `var`

이 장의 차례

6.1 바뀌는지는 낱말이 말한다 72
6.2 타입은 적거나, 값에게 맡긴다 74
6.3 값 없는 이름은 없다 74
6.4 지역이 사는 범위 75
6.5 `if` 는 값을 내지 않는다 76
6.6 흔한 실수 77
6.7 이 장의 문법 한눈에 79

먼저 알아야 할 것

3장 겉모습 · 가림(shadowing)이 금지된다
5장 op · op 안에 갇힌 변이는 순수로 친다

돌아보기

5장의 `running_total` 은 `var` 로 합을 쌓는데도 `fn` 이었다. 왜 허락되었는가?

> **답.** 순수함이 관측적이기 때문이다. `total` 은 op 안에서만 사는 지역이므로 그 변이를 호출자가 볼 수 없다. 호출자에게 보이는 쓰기(`mut` 매개변수)만 효과로 친다. 이 장은 그 지역을 짓는 두 낱말과, 지역이 사는 범위를 다룬다.

이 장의 필요성과 맥락

바뀌는 값은 읽는 사람에게 비싸다. 한 이름이 쓰이는 자리마다 "여기서는 무슨 값이지" 를 다시 물어야 하기 때문이다. Lowent 는 바뀌지 않는 이름과 바뀌는 이름을 **낱말**로 가르고, 값 없이 이름만 짓는 길을 막는다. 흐름(7장)에서 반복과 갈래를 쓰기 전에 그 안에서 값을 담아 둘 그릇을 먼저 정한다.

이 장이 끝나면

`let` 은 불변, `var` 는 가변이고 `set` 으로만 다시 넣는다는 것을 알게 된다. 타입을 적는 경우와 생략하는 경우, 값 없는 선언이 거절되는 이유(특히 `.5` 의 함정)를 익힌다. 지역이 블록 안에서만 살고, 살아 있는 바깥 이름을 안쪽에서 다시 쓸 수 없다는 것, 그리고 `if` 가 값을 내지 않는 문장이라는 것도 보게 된다.

이 장에서 답할 질문

1 `let` 으로 지은 슬라이스의 원소는 바꿀 수 있는가?
2 `if` 를 식으로 쓸 수 있으면 `var` 를 줄일 수 있지 않은가?

6.1 바뀌는지는 낱말이 말한다

op 본문에서 값을 담아 두는 이름이 **지역**이다. 두 가지뿐이다.

- `let <이름> <타입> be <식> .` — 불변. 한 번 정하면 바뀌지 않는다.
- `var <이름> <타입> be <식> .` — 가변. `set <이름> <식> .` 으로 다시 넣는다.

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
```

실행 결과

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

`total` 과 `i` 는 반복하며 바뀌므로 `var` 다. 바뀔 값만 `var` 로 짓고, 나머지는 `let` 으로 짓는 것이 기본이다. `let` 에 `set` 을 하면 거절된다.

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
```

실행 결과

```
$ 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
```

진단의 긴 설명에 따르면 한때는 `let` 과 `var` 가 같은 뜻이었다. 두 철자에 한 뜻이면 둘 중 하나는 아무것도 말하지 않는다. 지금 `let` 은 불변을 **약속**하고 컴파일러가 그 약속을 검사한다.

> **문.** `let` **으로 지은 슬라이스의 원소는 바꿀 수 있는가?**
>
> **답.** `let` 이 막는 것은 **이름에 다른 값을 다시 넣는 일**이다. 슬라이스의 원소에 쓸 수 있는지는 슬라이스의 타입이 정한다 — `mut slice` 면 원소에 쓸 수 있고 `slice` 면 읽기만 한다(9장). 이름의 불변과 가리키는 바이트의 불변은 서로 다른 질문이다.

6.2 타입은 적거나, 값에게 맡긴다

지역의 타입은 적을 수도 있고 생략할 수도 있다. 생략하면 값이 타입을 정한다.

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
```

실행 결과

```
$ 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)
```

`let b be add a 1 .` 에서 `a` 는 `u8` 이므로 `b` 도 `u8` 이다. 그래서 255 에 1 을 더하면 `u8` 에서 넘쳐 멈춘다. 값에서 타입을 정할 수 없으면 거절되고, 그때는 적어야 한다.

이 책은 대부분 타입을 적는다. 폭이 곧 넘침의 경계이므로(4장), 타입이 소스에 보이면 멈출 수 있는 자리가 함께 보이기 때문이다. 짧은 중간값이나 타입이 뻔한 자리에서만 생략한다.

6.3 값 없는 이름은 없다

`be` 뒤에는 값이 있어야 한다. 이름만 먼저 짓고 값을 나중에 주는 길은 이 언어에 없다.

examples/ch06/novalue.low

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

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

실행 결과

```
$ 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
```

이 예는 빈칸을 적은 실수가 아니다. 저자는 `.5` 라는 값을 적었다고 믿는다. 그러나 `.5` 는 부동소수 리터럴이 아니다 — 떨어진 마침표가 폼을 닫고, `be` 뒤에는 아무것도 남지 않는다. 옛 도구는 그 자리에 조용히 0 을 넣었고, 소스 어디에도 없는 값이 프로그램에 들어갔다. 지금은 거절되고, 진단이 이 함정을 이름으로 짚는다. 부동소수는 `0.5` 로 적는다.

> **흔한 오해. 초기화하지 않은 변수는 0 이다**
>
> C 에서 지역 변수를 초기화하지 않으면 값은 정해지지 않는다. 어떤 언어는 0 을 넣어 준다. Lowent 에는 둘 다 없다 — 초기화하지 않은 변수라는 것 자체가 없다. 값이 정해지지 않은 순간을 만들 수 없으므로, 그 순간에 읽는 결함도 없다.

6.4 지역이 사는 범위

지역은 자기가 지어진 블록이 끝날 때까지 산다. 블록이 끝나면 그 이름은 다시 쓸 수 있다.

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
```

실행 결과

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

두 `if` 블록은 각자 `doubled` 를 짓는다. 첫 블록이 끝나면 첫 `doubled` 는 사라지므로 둘째 블록에서 같은 이름을 다시 지어도 된다. 두 이름이 **동시에 살아 있는** 순간이 없기 때문이다.

반대로, 바깥에 살아 있는 이름을 안쪽 블록이 다시 지으면 거절된다.

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
```

실행 결과

```
$ 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
```

안쪽 블록에서 `t` 는 2 이고 블록을 나오면 다시 1 이다. 같은 글자가 줄에 따라 다른 값을 가리키면 읽는 사람이 틀린 값을 떠올린다. 3장에서 본 매개변수 가림과 같은 규칙이다 — 이름 공간은 평평하다.

6.5 `if` 는 값을 내지 않는다

`if` 는 문장이다. 식의 자리에 둘 수 없다.

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
```

실행 결과

```
$ 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
```

갈래마다 값을 정하려면 진단이 제안하는 대로 `var` 를 기본값으로 짓고 갈래 안에서 `set` 하거나, 갈래마다 `return` 한다. 값이 여러 경우 중 하나로 정해지는 모양이 되풀이되면 `match`(7장)가 더 맞는 도구다.

> **문. `if` 를 식으로 쓸 수 있으면 `var` 를 줄일 수 있지 않은가?**
>
> **답.** 줄어든다. 대신 식 안에 블록이 들어오고, 블록 안에서 `return` 이나 `break` 를 만나면 그 식이 어디로 빠져나가는지 따져야 한다. Lowent 는 식과 문장을 가르는 쪽을 골랐다. 값은 식이 만들고, 흐름은 문장이 바꾼다.

6.6 흔한 실수

반례. `set` 에서 이름을 잘못 적는다

examples/ch06/mistake_typo.low

```
module mistake_typo .
rem expect: E-IR-UNDEF

fn bump output u64 .
do
  var total u64 be 0 .
  rem ✘ 이름을 잘못 적었다 --- `totl` 은 지은 적이 없는 이름이다
  set totl (add total 1) .
  return total .
end
```

실행 결과

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

`set` 은 **이미 지은** `var` 에만 값을 다시 넣는다. 처음 보는 이름에 `set` 하면 새 변수를 만들어 주는 언어도 있지만, 여기서는 `E-IR-UNDEF` 로 거절한다. 오타가 새 변수가 되면 원래 변수는 영영 바뀌지 않은 채 조용히 틀린 답을 내기 때문이다. 짓는 것(`var`)과 바꾸는 것(`set`)을 다른 낱말로 가른 까닭이 이것이다.

반례. `=` 로 값을 넣는다

examples/ch06/mistake_equals.low

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

fn bump output u64 .
do
  var total u64 be 0 .
  rem ✘ `=` 는 이 언어의 기호가 아니다 --- 값을 다시 넣는 것은 `set` 이다
  total = add total 1 .
  return total .
end
```

실행 결과

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

대부분의 언어에서 `=` 는 대입이지만 수학에서는 같음이다. 한 기호가 두 뜻을 오가며 `if (x = 0)` 같은 결함을 낳아 왔다. Lowent 는 기호를 쓰지 않는다 — 짓는 것은 `let`/`var … be`, 바꾸는 것은 `set`, 같은지 묻는 것은 `eq` 다. `=` 는 아예 모르는 글자라 `E-CHAR` 가 나온다.

반례. 흔한 영어 낱말로 지역 이름을 짓는다

examples/ch06/mistake_localname.low

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

fn size input xs slice u8 . output u64 .
do
  rem ✘ `count` 는 내장 op 의 이름이다 --- 지역 이름으로 쓸 수 없다
  let count u64 be len xs .
  return count .
end
```

실행 결과

```
$ 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
```

`count` `len` `min` `max` `ok` `avg` 같은 이름은 이미 내장 op 이다. 이름 공간이 평평하므로 지역이 그 이름을 가져가면 `f count data` 가 "count 를 부른 값" 인지 "지역 count" 인지에 따라 문장이 다르게 묶인다. 같은 글자가 다른 나무가 되는 자리라서 `E-NAME-BUILTIN` 으로 막는다. `n` `total` `size` 처럼 짓는다.

반례. 블록 안에서 지은 지역을 블록 밖에서 읽는다

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` 은 위 블록 안에서만 산다 --- 블록 밖에서 읽었다
  return big .
end
```

실행 결과

```
$ 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` 은 `if` 블록 안에서 지었으므로 블록이 끝나면 사라진다. 그 이름을 블록 밖에서 읽으면 `E-NAME-SCOPE` 로 거절한다 — 전에는 블록에 들어가지 않은 길에서 **조용히 0 을** 주었다. 소스 어디에도 없는 값이다. 블록 밖에서도 쓸 값은 블록 **앞에서** 짓는다.

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 블록 밖에서도 쓸 값은 블록 앞에서 짓고, 블록 안에서는 set 만 한다
  var big u64 be 0 .
  if gt a 3 . do
    set big (mul a 10) .
  end
  return big .
end
```

실행 결과

```
$ 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
```

이제 0 은 소스에 적힌 기본값이다. 같은 답이지만 **어디서 온 0 인지**가 보인다.

6.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`let x T be e .`	바뀌지 않는 이름	기본은 불변 — 바뀌는 값만 따로 보이게
`var x T be e .`	바뀔 수 있는 이름	바뀔 것을 짓는 순간에 밝힌다
`set x e .`	`var` 에 새 값을 넣는다	짓기와 바꾸기를 다른 낱말로 — 오타가 새 변수가 되지 않게
`let x be e .`	타입을 값에게 맡긴다	짧은 중간값에만 — 폭이 곧 넘침의 경계라서
`be` 뒤의 값	반드시 있어야 한다	초기화하지 않은 변수라는 것이 없다
블록의 끝	그 안에서 지은 지역이 사라진다	이름은 쓰이는 곳 가까이 산다
안쪽에서 바깥 이름 다시 짓기	거절(`E-NAME-SHADOW`)	같은 글자는 한 값만 가리킨다
`eq a b`	같은가 묻기	`=` 가 없다 — 대입과 같음이 섞이지 않는다

표 6.1 — 지역의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

지역은 `let`(불변)과 `var`(가변) 둘이고 `set` 은 `var` 에만 한다. 타입은 적거나 값에게 맡길 수 있지만, `be` 뒤의 값은 반드시 있어야 한다. 지역은 자기 블록이 끝날 때까지 살고, 살아 있는 이름은 안쪽에서도 다시 지을 수 없다. `if` 는 값을 내지 않는 문장이다.

7 흐름 — 갈래, 되풀이, 빠져나가기

이 장의 차례

7.1 조건과 되풀이 80
7.2 `guard` — 조건을 사실로 바꾼다 82
7.3 모든 길이 값을 돌려준다 83
7.4 `match` — 경우를 빠짐없이 84
7.5 `panic` 은 효과다 86
7.6 흔한 실수 86
7.7 이 장의 문법 한눈에 91

먼저 알아야 할 것

4장 수 · 참거짓은 수가 아니다
6장 지역 · `if` 는 값을 내지 않는 문장이다

돌아보기

6장에서 `let x u64 be if gt a 1 . 5 else 6 .` 은 왜 거절되었고, 갈래마다 값을 정하려면 어떻게 적는가?

> **답.** `if` 는 값을 내지 않는 문장이라서 식의 자리에 둘 수 없다(`E-IF-VALUE`). 갈래마다 값을 정하려면 `var` 를 기본값으로 짓고 갈래 안에서 `set` 하거나, 갈래마다 `return` 한다. 이 장은 그 갈래 자체와, 되풀이와 빠져나가기의 규칙을 다룬다.

이 장의 필요성과 맥락

흐름의 문장은 어느 언어에나 있어서 새로 배울 것이 없어 보인다. 그러나 Lowent 의 흐름 문장에는 **검사**가 붙어 있다. 조건은 참거짓이어야 하고, `guard` 의 `else` 는 반드시 떠나야 하며, 값을 돌려주는 op 은 모든 길에서 돌려주어야 하고, `match` 는 모든 경우를 덮어야 한다. 이 검사들은 "어떤 길로 가면 값이 없다" 는 결함을 번역할 때 없앤다. 데이터(제3부)로 넘어가기 전에 흐름의 약속을 먼저 세운다.

이 장이 끝나면

`if … else`, `while`, `for`, `break`·`continue` 를 쓰는 법을 익힌다. `guard` 가 조건을 **그 아래 코드에 대한 사실**로 바꾸는 방식과, `else` 가 떠나지 않으면 왜 거절되는지 알게 된다. 모든 길이 값을 돌려주어야 한다는 규칙, 수와 범위를 가르는 `match` 와 그 망라 검사, 그리고 `panic` 이 효과라는 것도 보게 된다.

이 장에서 답할 질문

1. `for` 로 0 부터 `n` 까지 세는 반복은 어떻게 적는가?
2. `match` 가 `if` 사슬보다 나은 점은 망라 검사 말고 무엇인가?

7.1 조건과 되풀이

`if` 는 조건이 참일 때 블록을 실행하고, `else` 로 거짓일 때의 블록을 적는다. `while` 은 조건이 참인 동안 되풀이하고, `for` 는 슬라이스의 원소를 차례로 훑는다. 조건은 모두 `bool` 이어야 한다.

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
```

실행 결과

```
$ 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` 은 0 부터 `n` 미만까지 세면서 5 보다 큰 수만 센다. `while lt i n . do … end` 의 모양을 보면, 조건도 폼이므로 마침표로 닫고 그 뒤에 몸을 여는 `do` 가 온다.
- `first_zero` 는 0 을 만나면 `break` 로 되풀이를 벗어난다. 끝까지 없으면 길이를 돌려준다.
- `odd_sum` 은 `for x xs do … end` 로 원소를 훑고, 짝수면 `continue` 로 다음 원소로 넘어간다. `for` 의 이름 `x` 는 슬라이스의 원소 타입(`u8`)이고 블록 안에서만 산다.

  `for x in xs` 처럼 `in` 을 적으면 `E-VOCAB-REMOVED` 로 거절된다. 훑을 대상은 이름 바로 뒤에 온다.

> **문.** `for` 로 0 부터 `n` 까지 세는 반복은 어떻게 적는가?

> **답.** `for` 는 슬라이스를 훑는 도구라서 수의 범위를 직접 받지 않는다. 수를 세는 반복은 `while` 과 `var` 로 적는다. 세는 반복을 한 가지 모양으로만 적게 하면, 반복 변수의 범위를 컴파일러가 알아보기 쉽고 경계 검사를 지우는 분석도 단순해진다. 슬라이스를 걸러 세거나 모으는 일은 `pipe`(24장)가 맡는다.

7.2 `guard` — 조건을 사실로 바꾼다

`guard <조건> . else <빠져나감> .` 은 조건이 참이 아니면 **그 자리에서 떠난다.** 떠나는 문장은 `return`·`break`·`continue`·`panic` 이다.

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
```

실행 결과

```
$ 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
```

`head_or_zero` 의 `guard` 를 지난 코드는 슬라이스가 비지 않은 세계에서만 산다. 그래서 `index data 0` 이 안전하다. `grade` 는 `guard` 로 범위 밖을 먼저 걸러 내고, 그 아래에서 `if … end else do … end` 로 갈래를 나눈다.

`guard` 가 `if not` 의 다른 이름이 아닌 까닭은 `else` 가 반드시 떠나야 한다는 데 있다. 떠나지 않으면 거절된다.

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
```

실행 결과

```
$ 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`
```

`else` 가 떠나지 않고 아래로 흘러내리면, `guard` 아래 코드는 `n` 이 5 이하라고 믿지만 실제로는 그렇지 않다. 믿을 수 없는 사실을 믿게 두느니 번역을 거절한다. 떠나지 않을 일이면 `if` 를 쓴다.

> **흔한 오해. `guard` 는 코드를 짧게 하는 문법 설탕이다**
>
> 짧아지는 것은 부수 효과다. `guard` 의 본뜻은 **컴파일러와 읽는 사람에게 사실 하나를 건네는 것**이다. 컴파일러는 `guard` 를 지난 뒤 그 조건을 참으로 알고 경계 검사를 지우는 데 쓴다. 같은 뜻을 `if … do return … end` 로 적어도 동작은 같지만, 떠남이 문법으로 보장되지는 않는다.

7.3 모든 길이 값을 돌려준다

값을 돌려주는 op 은 **모든 길**에서 값을 돌려주고 끝나야 한다.

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
```

실행 결과

```
$ 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
```

`a` 가 0 이하인 길에는 `return` 이 없다. 옛 도구는 그 길에서 조용히 0 을 돌려주었다. 소스 어디에도 없는 값이다. 지금은 거절된다. 값을 돌려주고 끝나는 것으로 인정되는 문장은 `return`, 두 갈래가 모두 돌려주는 `if … else`, 모든 갈래가 돌려주는 `match` 다. `while·for·guard` 는 빠져나가는 길이 있으므로 그 자체로 돌려주는 것으로 보지 않는다. 그래서 반복 뒤에는 늘 `return` 이 온다.

돌려줄 값이 없는 op(`output void`)은 `return` 없이 몸의 끝까지 가도 된다. 끝나는 자리가 곧 돌아가는 자리다.

7.4 `match` — 경우를 빠짐없이

`match` 는 값을 경우별로 가른다. `if` 사슬과 다른 점은 **모든 경우를 다뤄야** 한다는 것이다.

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
```

실행 결과

```
$ 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` 는 `lo to hi` 범위(양 끝 포함)로 가른다. 정수는 경우가 많으므로 나머지를 덮는 `case _ .` 가 필요하다.
- `half` 는 두 범위가 `u8` 의 0 … 255 를 빈틈 없이 덮으므로 `_` 가 없어도 된다.
- `big` 의 `case y when gt y 100 .` 은 값 전체를 `y` 에 묶고 가드로 더 좁힌다. 가드가 붙은 갈래는 거짓일 수 있으므로 망라에 세지 않는다. 그래서 `_` 가 여전히 필요하다.

  빈틈이 있으면 거절된다.

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
```

실행 결과

```
$ 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 이 어느 갈래에도 들지 않는다. 같은 경우를 두 번 적거나 `_` 뒤에 갈래를 두면 `E-MATCH-REDUNDANT` 로 거절된다. 경고가 아니라 오류다 — 한 번도 실행될 수 없는 갈래는 대개 결함을 숨긴다.

> **문. `match` 가 `if` 사슬보다 나은 점은 망라 검사 말고 무엇인가?**
>
> **답.** 나중에 경우가 늘어날 때 드러난다. 열거형에 변형을 하나 더하면, 그 열거형을 가르는 모든 `match` 가 번역에서 멈춰 고쳐야 할 자리를 알려 준다(10장). `if` 사슬은 새 경우를 조용히 마지막 `else` 로 흘려보낸다. 또 `--ir` 로 보면 도구가 갈래 수와 최악 비교 횟수 같은 디스패치 비용을 말해 준다.

7.5 `panic` 은 효과다

`panic` 은 프로그램을 즉시 멈춘다. 되돌릴 수 없는 멈춤이므로 효과이고, 쓰는 op 은 `effects panic` 을 적는 `proc` 이어야 한다.

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
```

실행 결과

```
$ 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)
```

VM 은 `E-VM-PANIC` 을 내며, 이것이 **계약 위반이 아니라는** 것을 분명히 한다 — 코드가 멈추기로 고른 것이지 약속을 어긴 것이 아니다. 순수한 `fn` 은 `panic` 을 쓸 수 없다. 다만 넘침이나 계약 위반으로 처리기가 일으키는 멈춤은 `fn` 에서도 일어난다. 그것은 op 이 한 일이 아니라 처리기가 약속을 지키게 한 일이다.

`panic` 은 복구할 수 없는 상황에만 쓴다. 호출자가 다룰 수 있는 실패는 값으로 돌려준다 (11장). 둘을 가르는 기준은 17장이 다룬다.

7.6 흔한 실수

> **반례. `for x in xs` 로 적는다**
>
> **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 ✘ `in` 을 적었다 --- `for` 는 `for <이름> <슬라이스> do` 다
  for x in xs do
    set t (add t (widen u64 x)) .
  end
  return t .
end
```

실행 결과

```
$ 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` 는 `for <이름> <슬라이스> do` 다. 훑을 대상이 이름 바로 뒤에 오고, 몸의 시작은 `do` 가 알리므로 `in` 이 전할 것이 없다. 같은 뜻에 표시를 하나 더 두지 않으려고 없앴다. 고치는 법: `for x xs do`.

반례. `elif` 로 갈래를 잇는다

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` 는 낱말이 아니다 --- 이어지는 갈래는 `end else if … do` 다
  end elif lt n 80 . do
    return 1 .
  end
  return 2 .
end
```

실행 결과

```
$ 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`·`else if`: 는 언어마다 다르다. Lowent 는 이미 있는 낱말을 이어 붙인다 — 앞 블록을 `end` 로 닫고 `else if` 를 붙인다. `elif` 는 낱말이 아니므로 `elif lt n 80 .` 이 따로 선 문장으로 읽히고, 그 뒤의 `do … end` 는 여는 머리가 없는 블록이 된다(`E-BLOCK-NOHEAD`) — `do … end` 는 늘 `if`·`while`·`fn` 같은 머리가 연다.

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 갈래를 이을 때는 앞 블록을 `end` 로 닫고 `else if` 를 붙인다
  end else if lt n 80 . do
    return 1 .
  end else do
    return 2 .
  end
end
```

실행 결과

```
$ 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
```

마지막 갈래는 `end else do … end` 이다. 갈래가 셋 이상이고 모두 한 값을 가르는 것이면 `match` 가 더 알맞다.

반례. C 처럼 블록 안에 `else` 를 둔다

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 ✘ C 처럼 블록 안에 `else` 를 한 줄로 두었다 --- `end else do` 로 앞 블록을 닫아야
한다
  else
    set r 2 .
  end
  return r .
end
```

실행 결과

```
$ 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
```

C 나 몇몇 언어는 `if … { … } else { … }` 처럼 `else` 가 앞 블록 뒤에 붙는다. Lowent 에서 `else` 는 앞 블록을 `end` 로 닫은 **뒤에** 온다(`end else do`). 블록 안에 `else` 를 한 줄로 두면 `E-STMT-ELSE` 로 거절한다 — 전에는 번역이 통과하고 실행에서야 `E-VM-UNSUP` 으로 멈췄으며, 네이티브 빌드는 그 op 을 아예 뺐다.

반례. 조건에 `<` 같은 기호를 쓴다

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 ✘ `<` 는 이 언어의 기호가 아니다 --- 비교는 `lt i n` 이다
  while i < n . do
    set i (add i 1) .
  end
  return i .
end
```

실행 결과

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

비교는 낱말이다 — `lt`(작다)·`le`(작거나 같다)·`gt`·`ge`·`eq`·`ne`. `<` 는 이 언어가 모르는 글자라 `E-CHAR` 가 난다. 기호의 우선순위를 외우지 않아도 되게 하려는 선택이고, 긴 산술에는 `expr` 섬이 있다(8장). 고치는 법: `while lt i n . do`.

반례. `continue` 가 증가를 건너뛴다

`while` 로 세면서 몸 가운데서 `continue` 하면, 그 아래에 둔 `set i (add i 1) .` 도 함께 건너뛴다.

```
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
```

첫 짝수에서 `i` 가 더 이상 오르지 않아 반복이 끝나지 않는다. 번역도 실행 검사도 이것을 잡지 못한다 — 멈추지 않는 것은 넘침이 아니기 때문이다. 증가를 몸의 **맨 앞**으로 옮기거나(그러면 색인에는 증가 전의 값을 따로 담는다), 원소를 훑는 일이면 처음부터 `for x xs do` 를 쓴다. `for` 는 다음 원소로 넘어가는 일을 언어가 맡으므로 이 결함이 생길 자리가 없다.

나머지를 받는 자리는 두 가지로 적을 수 있다. `case _ .` 와 마지막 `else` 는 같은 일을 한다.

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 나머지를 `else` 로 받는다 --- `case _ .` 와 같은 자리다
    else do
      return 99 .
    end
  end
end
```

실행 결과

```
$ 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
```

`else` 뒤에는 갈래를 더 둘 수 없다 — 이미 다 받았으므로 그 뒤의 갈래는 한 번도 돌지 않고, `E-MATCH-REDUNDANT` 로 거절된다.

흔한 오해. `match` 의 갈래도 C 의 `switch` 처럼 아래로 흘러간다

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 맞은 갈래 하나만 실행된다 --- 다음 갈래로 흘러가지 않으므로 `break` 가 필요 없다
    case 1 . set score (add score 10) .
    case 2 . set score (add score 20) .
    case _ . set score (add score 1) .
  end
  return score .
end
```

실행 결과

```
$ 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
```

맞은 갈래 **하나만** 실행되고 다음 갈래로 흘러가지 않는다. `break` 를 적는 버릇도, 빠뜨려서 생기는 결함도 없다. 두 경우에 같은 일을 하려면 갈래를 `or` 로 묶는다(11장).

7.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`if c . do … end`	조건이 참이면 블록	조건도 폼이라 마침표로 닫고, 몸은 `do` 로 연다
`if c . do … end else do … end`	둘 중 하나	여는 말과 닫는 말이 늘 짝을 이룬다
`… end else if c2 . do … end`	갈래 잇기	새 낱말(`elif`) 없이 있는 낱말을 잇는다
`while c . do … end`	조건이 참인 동안	수를 세는 반복은 이 모양 하나
`for x xs do … end`	슬라이스의 원소를 차례로	다음 원소로 넘어가는 일을 언어가 맡는다
`break .` · `continue .`	반복에서 나가기 · 다음 회차로	흐름을 바꾸는 문장
`guard c . else return … .`	조건이 아니면 떠난다 — 지난 뒤엔 조건이 사실	`else` 가 반드시 떠나야 한다
`return e .`	값을 돌려주고 끝낸다	값을 내는 op 은 모든 길에서
`match v do case … . 문장 … end`	경우별로 가르기	빠짐없이, 겹침 없이 — 흘러내림이 없다
`case 1 to 9 .` · `case _ .` · `case y when c .`	범위 · 나머지 전부 · 가드가 붙은 갈래	정수는 경우가 많아 `_` 가 흔히 필요하다
`panic "…" .`	되돌릴 수 없는 멈춤(효과)	`effects panic` 을 적는 `proc` 에서만

표 7.1 — 흐름의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

조건은 `bool` 이어야 한다. `while` 은 조건으로, `for` 는 슬라이스로 되풀이하고 `break`·`continue` 로 흐름을 바꾼다. `guard` 의 `else` 는 반드시 떠나고, 지난 뒤에는 조건이 사실이 된다. 값을 돌려주는 op 은 모든 길에서 돌려주어야 한다. `match` 는 빠짐없이, 겹침 없이 경우를 덮어야 한다. `panic` 은 효과다.

8 식 — 전위 표기와 expr 섬

이 장의 차례

8.1 전위 표기에는 외울 것이 없다 92
8.2 expr 섬의 우선순위표 94
8.3 섬 안에서 하지 못하는 것 94
8.4 단락 평가로 조건을 지킨다 95
8.5 번역할 때 계산되는 식 96
8.6 흔한 실수 97
8.7 이 장의 문법 한눈에 100

먼저 알아야 할 것

3장 겉모습 · 이름이 먼저, 인자가 뒤
4장 수 · 비트 연산은 낱말로 적는다
7장 흐름 · `match` 는 모든 경우를 덮어야 한다

돌아보기

3장은 `expr a lt b lt c` 를 왜 거절했는가? 진단은 무엇으로 고치라고 했는가?

> **답.** 비교를 이어 쓰면 수학에서는 "a < b 이고 b < c" 로 읽히지만 많은 언어에서는 `(a < b) < c` 로 읽힌다. 같은 글자가 사람과 기계에게 다른 뜻이 되므로 `E-EXPR-CHAIN` 으로 거절했다. 진단은 `expr (a lt b) and (b lt c)` 로 적으라고 했다. 이 장은 그 섬 안의 규칙 전체를 다룬다.

이 장의 필요성과 맥락

식은 가장 많이 쓰는 문법이다. Lowent 의 식은 기본이 전위 표기이고 중위 표기는 `expr` 섬 안에서만 허락된다. 이 두 겹 구조를 제대로 알아야 괄호가 겹겹이 쌓인 전위 식을 읽을 수 있고, 섬 안에서 무엇이 허락되지 않는지를 알아 헤매지 않는다. 제2부의 마지막 장으로, 앞의 장들에서 조금씩 본 식의 규칙을 한곳에 모은다.

이 장이 끝나면

전위 표기에 우선순위가 없는 이유와, `expr` 섬의 짧은 우선순위표를 알게 된다. 섬 안에서 op 을 부를 때 괄호가 필요하고 단항 연산자가 없다는 것, 비교와 `and·or` 를 섞을 때 괄호로 묶는다는 것을 익힌다. `and·or` 의 단락 평가로 안전한 조건을 적는 법과, 번역할 때 계산되는 `size_of·comptime` 도 보게 된다.

이 장에서 답할 질문

1 VM 은 `in_range(5, 1, 9) = 1` 이라고 보여 준다. `bool` 이 수가 된 것인가?

8.1 전위 표기에는 외울 것이 없다

전위 표기에서 연산은 이름이 먼저 오고 인자가 뒤에 온다. 인자가 폼이면 괄호로 감싼다. 이 모양에는 우선순위가 없다. 무엇이 먼저 계산되는지는 괄호가 전부 말한다.

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
```

실행 결과

```
$ 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
```

`score_prefix` 의 `add a (mul b 2)` 와 `score` 의 `expr a + b * 2` 는 같은 식이다. 두 op 의 답이 같고, 섬은 전위로 번역되므로 실행 비용도 같다. 섬은 **표기의 투영**일 뿐 다른 연산이 아니다.

기호 대신 낱말을 쓰는 이유도 같은 곳에 있다. `^` 는 어떤 언어에서 거듭제곱이고 어떤 언어에서 배타적 논리합이다. `bit_xor` 는 어디서 읽어도 한 뜻이다. 그리고 낱말은 소리 내어 읽을 수 있다.

8.2 expr 섬의 우선순위표

섬 안에서만 우선순위가 있고, 표는 이것이 전부다.

단계	연산	결합	같은 뜻의 전위
5	`( )` 묶음	—	묶음 표시일 뿐 값이 아니다
4	`* /`	왼쪽	`mul` · `div`
3	`+ -`	왼쪽	`add` · `sub`
2	`eq ne lt le gt ge`	이어 쓸 수 없다	같은 이름의 전위 연산
1b	`and`	왼쪽	`and`(단락 평가)
1a	`or`	왼쪽	`or`(단락 평가)

표 8.1 — `expr` 섬 안의 우선순위(강한 것이 위)

`grouped` 의 `expr (a + b) * 2` 처럼 괄호로 차례를 바꿀 수 있다. 표에 없는 연산 — 나머지 `mod`, 비트 연산, `min`·`max` — 은 섬 안에서도 전위로 적고 괄호로 부른다. `spread` 의 `expr (max a b) - (min a b)` 가 그 모양이다.

`in_range` 는 비교 두 개를 `and` 로 묶었다. 표로는 비교가 `and` 보다 강하므로 괄호가 없어도 될 것 같지만, 이 판의 컴파일러는 `expr lo le x and x le hi` 를 `E-TYPE-LOGICAL` 로 거절한다. 비교를 `and`·`or` 와 섞을 때는 비교마다 괄호로 묶는다. 읽는 사람도 표를 떠올릴 필요가 없어진다.

> **문. VM 은 `in_range(5, 1, 9) = 1` 이라고 보여 준다. `bool` 이 수가 된 것인가?**
>
> **답.** 아니다. VM 의 결과 줄이 `bool` 을 0 과 1 로 **보여 줄** 뿐이다. 프로그램 안에서 `bool` 은 수와 섞이지 않는다(4장). 네이티브로 내보낸 C 에서도 `bool` 은 한 바이트의 0 또는 1 로 표현된다.

8.3 섬 안에서 하지 못하는 것

섬 안은 중위 연산자의 세계다. 그래서 두 가지가 없다.

첫째, 괄호 없는 op 부름. `twice a + 1` 은 `twice (a + 1)` 인지 `(twice a) + 1` 인지 알 수 없다.

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
```

실행 결과

```
$ 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
```

진단이 예를 들어 주듯, 섬 안의 부름은 `(twice a) + 1` 처럼 괄호로 묶는다.

둘째, 단항 연산자. 섬에는 왼쪽 피연산자가 없는 연산자가 없다.

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
```

실행 결과

```
$ 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)
```

부호를 뒤집으려면 `neg a` 를 전위로 적거나, 섬 안이라면 `(neg a)` 로 부르거나 `expr 0 - a` 로 적는다. `not b` · `bit_not x` 도 같다.

> **흔한 오해. 섬이 작은 것은 아직 덜 만들어서다**
>
> 섬의 크기는 의도다. 중위 표기의 값은 **아무도 우선순위를 찾아보지 않아도 될 때만** 성립한다. 표의 연산은 학교에서 배운 셈과 같아서 모두가 같게 읽는다. 비트 연산이나 단항 연산을 넣으면 읽기 쉬워지는 것이 아니라 찾아볼 것이 늘어난다. 섬은 앞으로도 커지지 않는다고 명세가 적어 두었다.

8.4 단락 평가로 조건을 지킨다

`and` 와 `or` 는 앞쪽만으로 답이 정해지면 뒤쪽을 계산하지 않는다. 이것을 이용하면 뒤쪽 식이 안전하게 계산될 조건을 앞쪽에 둘 수 있다.

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
```

실행 결과

```
$ 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` 는 빈 슬라이스에서 `index xs 0` 을 계산하지 않는다. 앞쪽 `gt (len xs) 0` 이 거짓이면 거기서 답이 거짓으로 정해지기 때문이다. `ratio_ok` 도 분모가 0 이면 나눗셈을 하지 않는다. 이 모양은 `guard` 를 쓰기엔 작은 조건에 알맞다. 조건이 op 전체에 대한 사실이라면 `guard` 가 낫다 (7장).

8.5 번역할 때 계산되는 식

어떤 식은 실행하기 전에 값이 정해진다.

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
```

실행 결과

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

`size_of u32` 는 그 타입 값 하나의 바이트 수를 번역할 때 준다. 타입을 매개변수로 받는 코드가 가장 넓은 폭을 가정하지 않고 비용을 정직하게 계산할 수 있게 한다(22장).

`comptime (add 2 3)` 은 식을 번역할 때 계산하라는 표시다. `match` 의 가를 값이 번역 시점 상수이면 `match` 는 맞는 갈래 하나로 접히고 실행 중 비교는 사라진다. 그래도 접힌 다른 갈래들은 타입 검사를 받는다. C 의 `#ifdef` 로 지운 코드가 검사도 받지 않는 것과 다르다. 빌드 설정을 읽는 `config <옵션>` 도 번역 시점 상수로 쓰인다(31장).

8.6 흔한 실수

비교(단계 2)가 `and`(단계 1b)보다 강하므로, 괄호 없이 섞어도 뜻은 하나다.

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 비교가 `and` 보다 강하므로 괄호 없이 섞어도 뜻이 하나다
  return expr lo le x and x le hi .
end
```

실행 결과

```
$ 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
```

그래도 괄호를 적으면 읽는 사람이 표를 떠올리지 않아도 된다. 아래가 같은 일을 괄호로 적은 것이다.

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
```

실행 결과

```
$ 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
```

반례. 나머지를 `%` 로 적는다

examples/ch08/mistake_percent.low

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

```
fn last_digit input a u64 . output u64 .
do
  rem ✘ `%` 는 섬의 연산자가 아니다 --- 나머지는 `mod` 를 괄호로 부른다
  return expr a % 10 .
end
```

실행 결과

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

섬의 연산자는 표에 있는 것이 전부다. `%` 는 C 에서는 나머지지만 언어마다 부호 규칙이 달라(음수의 나머지) 같은 기호가 다른 답을 낸다. Lowent 의 나머지는 이름 `mod` 하나이고 부호는 나누는 수를 따른다(4장). 섬 안에서는 괄호로 부른다.

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 표에 없는 연산은 섬 안에서도 전위로 적고 괄호로 감싼다
  return expr (mod a 10) + k .
end
```

실행 결과

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

반례. 거듭제곱을 ^ 로 적거나 정수에 pow 를 쓴다

`^` 는 섬의 연산자가 아니다(E-CHAR) — 어떤 언어는 거듭제곱, 어떤 언어는 배타적 논리합이라서 넣지 않았다. 정수의 제곱은 `mul a a` 로 적는다. 그리고 `pow` 는 **부동소수**의 거듭제곱이다. 이 판의 도구는 정수에 쓴 `pow` 를 거절하지 않고 틀린 값을 준다.

examples/ch08/mistake_pow.low

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

fn square input a u64 . output u64 .
do
  rem ✘ 정수에 `pow` 를 썼다 — 초월 함수는 부동소수 전용이다
  return pow a 2 .
end
```

실행 결과

```
$ 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
```

정본은 `pow·sqrt·sin·cos·exp·log·fmod` 를 **부동소수 전용**으로 정한다(§6.3.9). 정수를 주면 `E-TYPE-KIND` 로 거절된다. 2026-09-16 까지는 거절하지 않고 `square 5` 가 25 가 아니라 5 를 냈다 — 번역도 실행도 알리지 않는 조용한 오답이었다. 정수의 거듭제곱은 곱셈으로 적고, 부동소수로 셈하려면 `cast f64 n` 으로 먼저 옮긴다.

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` 는 부동소수에서 맞게 돈다
  return pow a 2.0 .
end

fn square_i input a u64 . output u64 .
  requires le a 1000 .
do
  rem 정수의 거듭제곱은 곱셈으로 적는다 — 이름이 하는 일을 말한다
  return mul a a .
end
```

실행 결과

```
$ 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
```

흔한 오해. 중첩된 부름에는 괄호가 반드시 있어야 한다

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 괄호가 없어도 `mul` 이 인자 둘을 받는다는 것으로 묶인다 --- 같은 13 이다
  return add 1 mul a b .
end
```

실행 결과

```
$ 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
```

괄호가 없어도 컴파일러는 **각 op 이 받는 인자 수**로 문장을 묶는다. `mul` 이 인자 둘을 받으므로 `add 1 mul a b` 는 `add 1 (mul a b)` 와 같다. 그래도 이 책은 괄호를 적는다. 인자 수를 외우지 않은 사람에게는 괄호가 묶음을 보여 주고, 지역 이름이 내장 op 과 같은 철자일 때(6장의 `count`) 묶음이 달라지는 사고도 막는다.

8.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`add a (mul b c)`	전위 표기 — 우선순위가 없다	괄호가 계산 차례를 다 말한다
`expr a + b * c`	중위 섬 — `* /` 가 `+ -` 보다 강하다	학교에서 배운 셈과 같은 것만 중위로
`expr (a + b) * c`	섬 안의 묶음	괄호는 값이 아니라 묶음 표시
`expr (lo le x) and (x le hi)`	비교를 논리와 섞기	비교마다 괄호 — 표를 떠올리지 않아도 된다
`expr (twice a) + 1`	섬 안에서 op 부르기	괄호가 없으면 어디까지가 부름인지 모른다(E-EXPR-APP)
`expr (mod a 10) + k`	표에 없는 연산(`mod`·비트·`min`)	섬은 커지지 않는다 — 전위로 부른다
`expr 0 - a` · `(neg a)`	부호 뒤집기	섬에 단항 연산자가 없다(E-EXPR-UNARY)
`and` · `or`	단락 평가 — 앞쪽으로 답이 정해지면 뒤를 계산하지 않는다	뒤쪽 식이 안전할 조건을 앞에 둔다
`size_of u32`	타입 하나의 바이트 수(번역할 때)	제네릭 코드가 비용을 정직하게 센다
`comptime (add 2 3)`	번역할 때 계산	접힌 갈래도 타입 검사는 받는다

표 8.2 — 식의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

식의 기본은 우선순위가 없는 전위 표기다. `expr` 섬은 사칙·비교·`and`·`or` 만 중위로 허락하고, 섬 안의 op 부름은 괄호로 묶으며 단항 연산자는 없다. 비교를 `and`·`or` 와 섞을 때는 비교마다 괄호로 묶는다. `and`·`or` 의 단락 평가로 뒤쪽 식을 지킬 수 있고, `size_of` 와 `comptime` 은 번역할 때 계산된다.

제3부 — 데이터

9 줄 — 배열과 슬라이스

이 장의 차례

9.1 시작과 길이를 함께 갖는다 102
9.2 쓰려면 mut slice 104
9.3 창을 좁힌다 105
9.4 계약이 경계 검사를 지운다 106
9.5 흔한 실수 108
9.6 이 장의 문법 한눈에 112

먼저 알아야 할 것

7장 흐름 · `for` 는 슬라이스를 훑고, `guard` 는 조건을 사실로 바꾼다
5장 op · 호출자에게 보이는 쓰기는 `proc` 이어야 한다

돌아보기

7장의 `head_or_zero` 에서 `index data 0` 이 안전했던 까닭은 무엇인가?

> **답.** 바로 앞의 `guard ge (len data) 1 . else return 0 .` 가 슬라이스가 비지 않았다는 사실을 그 아래 코드에 건넸기 때문이다. `guard` 를 지난 코드는 그 조건이 참인 세계에서만 산다. 이 장은 그 슬라이스가 무엇이고, 경계 검사가 언제 남고 언제 사라지는지를 다룬다.

이 장의 필요성과 맥락

C 에서 가장 값비싼 결함은 포인터가 가리키는 곳에 몇 개가 있는지를 사람이 따로 기억하다가 틀리는 데서 나온다. 버퍼 넘침이 그것이다. Lowent 에는 포인터가 없고, 여러 값을 가리키는 일은 **길이를 함께 가진** 슬라이스가 맡는다. 제3부의 첫 장이 슬라이스인 것은 문자열·버퍼·파일 내용 같은 거의 모든 데이터가 슬라이스로 오가기 때문이다.

이 장이 끝나면

`array n t` 와 `slice t` 의 차이, `len`·`index`·`for` 로 읽는 법, 범위 밖 접근이 멈춘다는 것을 알게 된다. `mut slice` 로 받아야 원소에 쓸 수 있다는 규칙과 `subslice` 로 창을 좁히는 법을 익힌다. 계약이 본문의 경계 검사를 지우는 원리와, 문자열 리터럴이 그대로 색인되는 표라는 것도 보게 된다.

이 장에서 답할 질문

1. 문자열은 따로 된 타입이 없는가?
2. 슬라이스 인자를 `--run` 에 줄 때 `[3,4,5]` 는 `slice u64` 에도 쓸 수 있는가?

9.1 시작과 길이를 함께 갖는다

`slice t` 는 타입 `t` 의 값이 연속으로 놓인 구간이다. 슬라이스 값은 **시작 주소와 길이**를 함께 갖는다. 그래서 “여기 몇 개가 있는가” 를 사람이 따로 기억할 필요가 없다.

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
```

실행 결과

```
$ 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` 는 원소 개수, `index data 0` 은 첫 원소다. 번호는 0 부터다.
- `last4` 의 `array 4 u8` 은 길이가 정확히 4 인 줄이다. 입력 자리의 `array` 는 길이가 4 인 슬라이스를 받는다는 뜻이고, 그 길이는 `requires eq (len xs) 4 .` 처럼 진입에서 검사된다.
- `for x xs do … end` 는 원소를 차례로 훑는다.
- `at` 에 길이 3 인 슬라이스와 5 를 주면 `E-VM-BOUNDS` 로 멈춘다. 남의 메모리를 읽지 않는다.

`array` 는 길이를 **먼저** 적는다. 차례를 뒤집으면 거절된다.

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
```

실행 결과

```
$ 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
```

진단에 따르면 옛 도구는 `array u8 4` 를 그냥 슬라이스로 읽고 길이를 조용히 버렸다. 적은 약속이 사라지는 자리였다. 지금 `array` 는 op 의 입력 자리에서만 쓰고, 출력·지역·구조체 칸에서는 `slice` 와 계약으로 적는다.

> **문. 문자열은 따로 된 타입이 없는가?**
>
> **답.** 없다. 문자열 리터럴 `"hello"` 는 `slice u8` 이다. 글자가 몇 개인지, UTF-8 로 올바른지 같은 질문은 타입이 아니라 라이브러리의 op 이 답한다(33장). 바이트와 글자를 같은 것으로 여기는 순간 생기는 결함을 피하려는 선택이다.

9.2 쓰려면 `mut slice`

슬라이스의 원소에 쓰려면 그 슬라이스가 `mut` 여야 한다. 그리고 호출자가 넘긴 바이트를 바꾸는 일은 호출자에게 보이므로 `proc` 이어야 한다(5장).

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
```

실행 결과

```
$ lowentc --run fill_two fill.low [0,0,5]
fill_two([10,20,5]) = 35
  arg0 (written) = [10,20,5]
```

VM 이 보여 주는 `fill_two([10,20,5])` 는 op 이 끝난 뒤 인자의 모습이다. 호출자의 바이트가 바뀌었다. `mut` 없이 쓰면 거절된다.

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
```

실행 결과

```
$ 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` 는 슬라이스의 **원소**에 대한 권한이다. 슬라이스를 담은 이름을 `let` 으로 지었는지와는 다른 질문이다 (6장). 원소에 쓸 수 있는 슬라이스는 한 번에 한 곳에서만 빌릴 수 있다는 규칙이 뒤따른다 (12장).

9.3 창을 좁힌다

`subslice s from to` 는 `s` 의 `from` 번부터 `to` 번 **앞**까지를 가리키는 새 슬라이스를 만든다. 바이트를 베끼지 않는다. 같은 바이트 위에 좁은 창을 하나 더 얹을 뿐이다.

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
```

실행 결과

```
$ 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` 은 양 끝을 뺀 가운데 원소들의 합을 구한다. `prefix_char` 는 문자열 리터럴을 그대로 색인한다. 리터럴이 `slice u8` 이기 때문이다.

> **실제 사례. `if` 사슬을 표 하나로**
>
> 색인에 따라 상수를 돌려주는 op 을 `if eq k 0 . do return 47 . end` 같은 사슬로 쓰기 쉽다. 그러면 방출된 C 는 비교와 분기를 글자마다 하나씩 갖는다. 문자열 리터럴을 색인하면 정적 표 하나와 색인 한 번이 된다. HTTP 요청을 읽는 벤치마크에서 이 사슬을 표 조회로 바꾸자 분기 예측 실패가 손으로 쓴 C 와 같은 수준이 되었고 실행 시간이 약 25% 줄었다. 답은 한 비트도 바뀌지 않았다.

examples/ch09/table.low

```
module table .
rem run: prefix_char 4
rem run: prefix_chain 4

rem 문자열 리터럴은 slice u8 이고 그대로 색인된다
fn prefix_char input k u64 . output u8 .
  requires lt k 11 .
do
  return index "/api/users/" k .
end

rem 같은 일을 if 사슬로 쓰면 글자마다 비교와 분기가 하나씩 생긴다
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
```

실행 결과

```
$ lowentc --run prefix_char table.low 4
prefix_char(4) = 47
$ lowentc --run prefix_chain table.low 4
prefix_chain(4) = 47
```

두 op 의 답은 같다. `--emit-c` 로 보면 `prefix_char` 는 문자열 표 하나와 색인 한 번이고, `prefix_chain` 은 비교와 `goto` 의 사슬이다. 다만 이 판에서는 `prefix_char` 의 색인 검사가 남는다 — 분석이 리터럴의 길이 11 을 계약 `lt k 11` 과 잇지 않기 때문이다.

9.4 계약이 경계 검사를 지운다

`index` 의 경계 검사는 증명되지 않은 자리에만 남는다. 길이 조건을 `requires` 로 적으면 검사는 op 진입에서 **한 번**만 일어나고, 본문의 색인 검사는 사라진다.

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
```

실행 결과

```
$ 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]
```

`requires le n (len a) .` 가 있으면, `while lt i n .` 안에서 `i` 는 언제나 `len a` 보다 작다. 컴파일러의 구간 분석이 그것을 알아내고 `index a i` 의 검사를 지운다. 이 논증의 규칙들은 Coq 로 증명되어 있고, 지운 자리마다 컴파일러가 남긴 산술 근거를 독립 검산기가 다시 따진다(41장).

주의할 것이 하나 있다. 개수 `n` 에는 `le` 를 쓰지만, **색인** 자체에는 `lt` 를 쓴다. `requires le i (len a) .` 는 `i = len a` 를 허락하고, 그것은 한 칸 밖이다.

> **흔한 오해. `len` 을 반복 조건에 두면 매번 비싸게 센다**
>
> `len s` 는 호출도 메모리 읽기도 아니다. 슬라이스 값 {시작, 길이} 의 길이 칸을 꺼낼 뿐이다. 원소에 써도 길이는 바뀌지 않는다. 길이가 바뀌는 길은 그 이름에 다른 슬라이스를 다시 담는 것 하나다. 오히려 길이를 미리 `let m u64 be len xs .` 로 담아 두었다가 `xs` 를 더 짧은 슬라이스로 바꾸면 `m` 은 **낡은 길이**가 되어 범위 밖으로 갈 수 있다. 미리 담는 것은 최적화가 아니라 의미의 선택이다.

examples/ch09/stale_len.low

```
module stale_len .
rem run: fresh [1,2,3,4]
rem trap: stale [1,2,3,4]

rem len 을 조건에 그대로 둔다 --- 매 바퀴 지금의 길이를 읽는다
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 길이를 미리 담았다 --- s 가 짧아져도 m 은 그대로다
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
```

실행 결과

```
$ 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` 는 매 바퀴 지금의 길이를 읽으므로 `s` 가 두 칸으로 줄면 반복도 거기서 멈춘다. `stale` 은 처음 길이 4 를 `m` 에 담아 두었으므로 셋째 바퀴에서 두 칸짜리 `s` 의 2 번 칸을 읽으려다 멈춘다. `len` 은 매번 지금 값을 읽고, 그것이 안전을 지킨다.

> **문. 슬라이스 인자를 `--run` 에 줄 때 `[3,4,5]` 는 `slice u64` 에도 쓸 수 있는가?**
>
> **답.** `--run` 의 대괄호 인자는 바이트의 나열이다. 그래서 이 책의 실행 예제는 슬라이스를 `slice u8` 로 받는다. `slice u64` 에 넘기면 바이트 수가 8 의 배수가 아니라서 VM 이 멈춘다. 넓은 원소의 슬라이스는 프로그램 안에서 만들어 넘기는 것이 보통이다.

9.5 흔한 실수

줄을 다룰 때의 실수는 거의 모두 **번호 하나 차이**에서 온다. C 라면 남의 메모리를 읽고 조용히 지나갔을 자리에서 Lowent 는 멈춘다.

> **반례. 반복 조건에 `le` 를 써서 한 칸을 더 돈다**
>
> **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 ✘ `le` 라서 i 가 3 일 때도 돈다 --- 길이 3 인 줄의 번호는 0, 1, 2 뿐이다
>   while le i (len xs) . do
>     set t (add t (widen u64 (index xs i))) .
>     set i (add i 1) .
> ```

```
  end
  return t .
end
```

실행 결과

```
$ lowentc --run total mistake_offbyone.low [1,2,3]
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

길이가 3 인 줄의 번호는 0, 1, 2 다. `le i (len xs)` 는 `i` 가 3 일 때도 참이라 네 번째 칸을 읽으려다 `E-VM-BOUNDS` 로 멈춘다. 번호가 0 부터 시작하므로 "개수보다 작다" 가 맞는 조건이다 — `while lt i (len xs) ..` 원소를 모두 훑는 일이면 번호를 쓰지 않는 `for x xs do` 가 이 실수를 아예 없앤다.

반례. `xs[0]` 처럼 대괄호로 읽는다

examples/ch09/mistake_cindex.low

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

fn first input xs slice u8 . output u8 .
do
  rem ✘ 대괄호 색인은 없다 --- `index xs 0` 으로 읽는다
  return xs[0] .
end
```

실행 결과

```
$ 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
```

대괄호 색인은 없다. 원소를 읽는 것은 `index xs 0`, 쓰는 것은 `set (index xs 0) v .` 다. 기호 대신 이름을 쓰면 읽기·쓰기·범위 밖 멈춤이 모두 같은 모양의 폼이 되고, 기호를 외울 것이 줄어든다.

반례. 빈 줄일 수 있는데 첫 원소를 읽는다

examples/ch09/mistake_empty.low

```
module mistake_empty .
rem trap: first []

fn first input xs slice u8 . output u8 .
do
  rem ✘ 빈 줄에는 0 번 원소가 없다
  return index xs 0 .
end
```

실행 결과

```
$ lowentc --run first mistake_empty.low []
== ir diagnostics (1) ==
0:0 E-VM-BOUNDS: slice index out of bounds (panic)
```

입력이 비었을 가능성을 잊기 쉽다. 번호 0 은 원소가 하나 이상일 때만 있다. 먼저 걸러 내고 읽는다.

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 비었으면 여기서 떠난다 --- 아래에서는 원소가 하나 이상이라는 것이 사실이다
  guard gt (len xs) 0 . else return fallback .
  return index xs 0 .
end
```

실행 결과

```
$ 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]
```

`guard` 를 지난 아래에서는 "비지 않았다" 가 사실이므로 `index xs 0` 이 안전하고, 컴파일러도 그 사실로 경계 검사를 지운다 (7장).

반례. `subslice` 의 시작과 끝을 뒤집는다

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 ✘ 시작(3)이 끝(1)보다 뒤다 --- `subslice s from to` 는 from ≤ to ≤ len 이어야 한다
  return len (subslice xs 3 1) .
end
```

실행 결과

```
$ 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` 는 `from ≤ to ≤ len s` 여야 한다. 뒤집으면 빈 창이 되는 것이 아니라 멈춘다 — 잘못 계산한 경계가 조용히 빈 결과로 바뀌면 그 결함은 한참 뒤에야 드러나기 때문이다.

흔한 오해. `subslice s 1 3` 은 1 번부터 3 번까지 세 칸이다

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 1 번부터 3 번 앞까지 --- 1 번과 2 번, 둘이다
  return len (subslice xs 1 3) .
end
```

실행 결과

```
$ 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]
```

끝 번호는 **포함하지 않는다**. `1` 번과 `2` 번, 두 칸이다. 이렇게 정하면 길이가 `to - from` 으로 바로 나오고, `subslice s 0 k` 와 `subslice s k (len s)` 가 겹치거나 빠지는 칸 없이 줄을 나눈다. 대부분의 언어가 같은 규칙(반열린 구간)을 쓰는 까닭이다.

반례. 문자열 리터럴을 `mut slice` 로 받아 고친다

examples/ch09/mistake_litwrite.low

```
module mistake_litwrite .
rem expect: E-TYPE-ARGMUT

rem ✘ 문자열 리터럴을 mut slice 로 받아 고친다 — 리터럴은 프로그램에 박힌 읽기 전용 바이
트다
fn poke output u8 . do
  let buf mut slice u8 . be "abc" .
  set (index buf 0) 65 .
  return index buf 0 .
end
```

실행 결과

```
$ 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
```

문자열 리터럴은 프로그램에 박힌 바이트라 고칠 자리가 아니다. 그래서 `mut` 로 선언한 이름에 묶는 것도, `mut` 자리에 넘기는 것도 `E-TYPE-ARGMUT` 로 거절된다. 2026-09-16 까지는 둘 다 통과했고, 실행하면 VM 은 65 를 네이티브는 97 을 돌려줬다 — 같은 프로그램이 두 답을 냈고, 표준 라이브러리가 그 바이트에 쓰면 네이티브는 세그폴트로 죽었다. 고칠 바이트는 `alloc_bytes` 나 호출자의 버퍼로 받고, 리터럴은 거기에 베껴 넣는다.

9.6 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`slice u8`	`u8` 이 연속으로 놓인 구간(시작 + 길이)	개수를 따로 들고 다니지 않게
`mut slice u8`	원소에 쓸 수 있는 구간	쓸 수 있는지가 타입에 보인다
`input xs array 4 u8 .`	길이가 정확히 4 인 줄을 받는다(입력 자리만)	길이를 먼저 적는다 — 진입에서 검사
`len xs`	원소 개수	길이 칸을 꺼낼 뿐, 비용이 없다
`index xs i`	`i` 번 원소(0 부터)	범위 밖이면 멈춘다 — 남의 메모리를 읽지 않는다
`set (index xs i) v .`	`i` 번에 쓰기	`mut slice` 와 `proc` 이어야 한다
`for x xs do … end`	원소를 차례로	번호 실수가 생길 자리가 없다
`subslice xs from to`	`from` 부터 `to` 앞까지의 창(베끼지 않음)	반열린 구간 — 길이는 `to - from`
`"hello"`	`slice u8` 인 리터럴 — 그대로 색인된다	문자열 타입이 따로 없다
`requires le n (len xs) .`	길이 조건을 계약으로	본문의 경계 검사가 지워진다

표 9.1 — 줄의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

슬라이스는 시작과 길이를 함께 갖는다. `len` 은 길이 칸을 꺼내고, `index` 는 범위 밖이면 멈추며, `for` 는 원소를 훑는다. `array n t` 는 길이를 먼저 적고 입력 자리에서 쓴다. 원소에 쓰려면 `mut slice` 와 `proc` 이 필요하다. `subslice` 는 베끼지 않고 창을 좁힌다. 길이 조건을 계약으로 적으면 본문의 경계 검사가 지워지고, 문자열 리터럴은 그대로 색인되는 표다.

10 묶음 — struct 와 enum

이 장의 차례

10.1 struct — 이름 붙은 칸의 모음 113
10.2 enum — 여럿 중 하나 115
10.3 갈래는 마침표로 닫는다 116
10.4 자기 자신을 품을 수 없다 117
10.5 흔한 실수 118
10.6 이 장의 문법 한눈에 122

먼저 알아야 할 것

7장 흐름 · match 는 모든 경우를 덮어야 한다
9장 줄 · 슬라이스는 길이를 함께 갖는다

돌아보기

7장에서 match 가 if 사슬보다 나은 점으로, 망라 검사 말고 무엇을 들었는가?

> **답.** 나중에 경우가 늘어날 때를 들었다. 열거형에 변형을 하나 더하면 그 열거형을 가르는 모든 match 가 번역에서 멈춰 고칠 자리를 알려 주지만, if 사슬은 새 경우를 조용히 마지막 else 로 흘려보낸다. 이 장에서 그 열거형을 직접 만든다.

이 장의 필요성과 맥락

수와 슬라이스만으로는 "점 하나", "선분 하나", "원이거나 사각형이거나 점" 같은 모양을 적을 수 없다. 이름 붙은 칸의 모음(struct)과 여럿 중 하나(enum)가 있어야 데이터가 문제의 모양을 닮는다. 특히 값을 지닌 enum 과 match 의 짝은 Lowent 에서 상속을 대신하는 주된 도구이고, 뒤에 나오는 option 과 result(11장), 트레이트(23장)가 모두 이 위에 선다.

이 장이 끝나면

struct 를 선언하고 make 로 만들고 field 로 읽는 법을 익힌다. 붙임 점이 없는 이유를 알게 된다. enum 의 갈래가 값을 지닐 수 있다는 것, match 로 갈래를 가르며 값을 이름에 묶는 법, 갈래를 마침표로 닫아야 하는 이유를 보게 된다. 자기 자신을 품는 타입이 거절되는 까닭과, 나무 구조를 번호로 잇는 법도 알게 된다.

이 장에서 답할 질문

1 칸 이름으로 to 나 in 을 쓸 수 있는가?
2 번호로 잇는 것은 포인터로 잇는 것보다 불편하지 않은가?

10.1 struct — 이름 붙은 칸의 모음

struct 는 이름 붙은 칸들의 모음이고, 칸마다 자기 타입이 있다. 선언의 몸은 do … end 이고 칸마다 마침표로 닫는다.

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
```

실행 결과

```
$ 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` 가 값을 만든다. 만들 때 **모든 칸을 채워야** 한다. 채우지 않은 칸이 조용히 0 이 되는 일은 없다.
- `field s stop x` 는 `s` 의 `stop` 칸, 그 안의 `x` 칸을 읽는다. 마디를 여럿 적으면 왼쪽부터 한 칸씩 내려간다.
- 칸의 타입이 다른 `struct` 여도 된다. `segment` 는 `point` 둘을 품는다.
- `moved` 는 칸 하나를 바꾼 **새 값**을 만들어 돌려준다. 원래 값은 그대로다.

`p.x` 같은 붙임 점은 없다. 점은 이미 모듈 이름(`allocs.byte_allocator`)과 갈래 이름(`shape.dot`)에 쓰인다. 값의 안을 보는 일까지 점으로 적으면, `a.b` 가 무엇인지는 `a` 가 값인지 모듈인지를 찾아본 뒤에야 정해진다. `field` 폼은 읽는 순간에 정해진다.

> **문. 칸 이름으로 `to` 나 `in` 을 쓸 수 있는가?**
>
> **답.** 쓸 수 없다. `to` 와 `in` 은 옛 문법에서 칸을 읽는 중위 표기(`a to b`)였고 지금은 없앤 낱말이다. 낱말과 같은 철자는 이름이 될 수 없어서 `E-VOCAB-REMOVED` 로 거절된다. 이 예제가 `from·to` 대신 `start·stop` 을 쓴 이유다.

`field` 는 읽는 자리이면서 쓰는 자리이기도 하다. `mut` 로 받은 값의 칸은 `set (field p x) 3 .` 으로 바꾼다. 읽기와 쓰기가 같은 철자다. 값을 빌려 바꾸는 규칙은 12장에서 다룬다.

10.2 `enum` — 여럿 중 하나

`enum` 은 여러 갈래 가운데 하나다. 갈래는 값을 지닐 수 있고, 지니는 값은 `<칸 이름> <타입>` 짝으로 적는다.

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
```

실행 결과

```
$ 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 .` 는 `r` 이라는 값 하나를, `rect w u32 h u32 .` 는 둘을 지닌다. `dot .` 은 아무것도 지니지 않는다.
- 값을 만들 때는 `shape.circle 2` 처럼 **갈래 이름**으로 적는다.
- `match` 의 `case rect w h .` 는 갈래가 `rect` 이면 지닌 두 값을 `w` 와 `h` 에 묶는다.

갈래가 몇 개든 `match` 는 모두 덮어야 한다. `dot` 을 빠뜨리면 거절된다.

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
```

실행 결과

```
$ 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
```

이 검사가 `enum` 의 가장 큰 값이다. 나중에 `triangle` 갈래를 더하면, `area` 와 `corners` 를 비롯해 `shape` 를 가르는 모든 `match` 가 번역에서 멈춘다. 고쳐야 할 자리를 사람이 기억할 필요가 없다.

`match` 를 쓰지 않고 갈래만 물을 때는 `isa n lit` 이 `bool` 을 준다.

10.3 갈래는 마침표로 닫는다

개행은 닫개가 아니다(3장). 그래서 갈래를 줄마다 적고 마침표를 빠뜨리면 여러 갈래가 한 갈래로 이어진다.

examples/ch10/enum_dot.low

```
module enum_dot .
rem expect: E-ENUM-DOT

enum color do
  red
  green
  blue .
end
```

실행 결과

```
$ 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` 가 한 갈래의 이름과 칸 짝으로 읽힐 뻔했다. 컴파일러가 거절하고, 갈래마다 `red .` 처럼 닫으라고 알려 준다.

> **흔한 오해. `enum` 은 C 의 열거형처럼 이름 붙은 정수다**
>
> C 의 `enum` 은 정수 상수에 이름을 붙인 것이고, 정수와 섞여 쓰인다. Lowent 의 `enum` 은 **타입**이다. 정수와 섞이지 않고, 갈래마다 다른 모양의 값을 지닐 수 있으며, 가를 때 모든 갈래를 덮어야 한다. Rust 의 `enum` 이나 ML 계열의 대수적 자료형에 가깝다.

10.4 자기 자신을 품을 수 없다

갈래가 자기 타입을 **값으로** 품으면 크기가 무한해진다.

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
```

실행 결과

```
$ 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)
```

`struct` 도 같다. 직접이든 다른 타입을 거쳐서든 자기를 칸으로 품으면 `E-STRUCT-CYCLE` 로 거절된다. 나무 같은 구조가 필요하면 진단의 제안대로 **번호**로 잇는다. 노드들을 슬라이스에 담고, 자식은 그 슬라이스의 번호로 가리킨다.

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
```

실행 결과

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

`add l u32 r u32 .` 의 `l`·`r` 은 노드 자체가 아니라 `nodes` 안의 번호다. `eval` 은 번호로 자식을 찾아 재귀한다. 크기가 정해지고, 번호를 따라가는 `index` 는 경계 검사를 그대로 받는다. 이런 노드 슬라이스를 만드는 저장소 — 영역 — 는 18장에서 다룬다.

> **문. 번호로 잇는 것은 포인터로 잇는 것보다 불편하지 않은가?**
>
> **답.** 한 가지는 불편하다. 노드를 지우고 그 번호를 다시 쓰면, 옛 번호를 들고 있던 쪽이 엉뚱한 노드를 본다. 표준 라이브러리의 `pool` 은 번호에 세대 수를 함께 담는 **세대 핸들**로 이 문제를 실행 중에 잡는다(34장). 대신 얻는 것이 크다. 번호는 복사해도 안전하고, 파일에 그대로 쓸 수 있으며, 노드들이 한 슬라이스에 붙어 있어 캐시에 잘 맞는다.

10.5 흔한 실수

> **반례. `make` 에서 칸 하나를 빠뜨린다**
>
> **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` 를 채우지 않았다 --- `make` 는 모든 칸을 채워야 한다
>   let p point be make point do x 1 . end .
>   return field p x .
> end
> ```

실행 결과

```
$ 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 의 구조체 초기화는 빠진 칸을 0 으로 채우고, 어떤 언어는 기본값을 넣는다. Lowent 는 `E-TYPE-FIELD` 로 거절한다. 빠진 칸이 의도였는지 실수였는지 소스에서 알 수 없기 때문이다. 0 이 필요하면 `y 0 .` 이라고 적는다 — 적힌 0 은 누가 봐도 의도다. 칸 이름을 잘못 적어도 같은 코드가 "그런 칸이 없다" 로 나온다.

반례. 값을 지니지 않은 갈래를 `타입.갈래` 로 넘긴다

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 값을 지니지 않은 갈래는 이름만 적는다
  return code green .
end

fn pick_qualified output u64 .
do
  rem ✘ 이 판에서는 `light.green` 으로 넘기면 실행 중에 멈춘다
  return code light.green .
end
```

실행 결과

```
$ lowentc --run pick_bare mistake_unitvariant.low
pick_bare() = 2
$ lowentc --run pick_qualified mistake_unitvariant.low
pick_qualified() = 2
```

값을 지니는 갈래는 `shape.circle 2` 처럼 타입 이름을 붙여 만들고, 아무것도 지니지 않는 갈래는 `green` 처럼 이름만 적는 것이 이 책의 버릇이다. 둘 다 같은 값이다 — `pick_bare` 와 `pick_qualified` 가 같은 2 를 낸다. 2026-09-16 까지 이 판의 도구는 값 없는 갈래를 `light.green` 으로 적으면 **다른 표현**(태그를 가진

레코드)으로 낮췄고, 그래서 실행 중에 `E-VM-TYPE` 으로 멈췄다. 지금은 둘이 같은 인덱스로 낮아진다. 짧은 쪽을 쓰되, 어느 열거형의 갈래인지 헷갈리는 자리에서는 타입 이름을 붙여도 된다.

반례. 구조체 둘을 `eq` 로 비교한다

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 ✖ 구조체 둘을 `eq` 로 비교했다 --- `eq` 는 수와 참거짓을 비교한다
  return eq p q .
end
```

실행 결과

```
$ 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` 는 수와 참거짓을 비교한다. 구조체 둘이 "같다" 는 것은 타입마다 뜻이 다르다 — 모든 칸이 같아야 하는지, 이름표 칸만 같으면 되는지. 그래서 `E-TYPE-KIND` 로 번역 때 거절한다. 같음의 뜻을 op 으로 적는다.

examples/ch10/eqstruct_fixed.low

```
module eqstruct_fixed .
rem run: same

struct point do
  x u64 .
  y u64 .
end

rem 무엇이 "같은" 것인지 칸마다 적는다 --- 그 뜻은 타입마다 다르다
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
```

실행 결과

```
$ lowentc --run same eqstruct_fixed.low
same() = 1
```

반례. 다른 이름에 담으면 복사본이 된다고 믿는다

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 ✘ 복사본을 만든다고 믿었다 --- 이 판에서 q 는 p 와 같은 자리를 가리킨다
  var q point be p .
  set (field q x) 99 .
  rem p 는 let 인데도 99 가 나온다
  return field p x .
end
```

실행 결과

```
$ lowentc --run copy_then_change mistake_alias.low
copy_then_change() = 1
```

`var q point be p .` 는 `p` 의 칸을 **베낀 새 값**을 만든다. 그래서 `q` 의 칸을 99 로 바꾸어도 `p` 는 1 그대로다. 소유가 없는 값은 베껴지고, 소유가 있는 값은 옮겨진다(19장). 2026-09-16 까지 이 판의 도구는 구조체를 담을 때 같은 자리를 가리키는 별칭을 만들어 `p` 까지 바뀌었다 — `let` 의 약속이 깨지는 자리였고, 지금은 하강이 칸을 베낀다.

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 새 값이 필요하면 make 로 새로 만든다 --- 칸을 하나씩 옮겨 적는다
```

```
  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
```

실행 결과

```
$ lowentc --run copy_then_change alias_fixed.low
copy_then_change() = 1
```

이 장 첫 예제의 `moved` 가 칸을 바꾸지 않고 새 값을 만들어 돌려주는 모양인 까닭도 같다. 값을 고치기보다 새 값을 만드는 편이 별칭 문제를 아예 피한다.

10.6 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`struct point do x u64 . y u64 . end`	이름 붙은 칸의 모음	칸마다 이름과 타입이 한 줄씩
`make point do x 1 . y 2 . end`	값 만들기 — 모든 칸을 채운다	빠진 칸이 조용히 0 이 되지 않게
`field p x` · `field s stop x`	칸 읽기 · 여러 마디 내려가기	붙임 점 없이 — 읽는 순간 뜻이 정해진다
`set (field p x) 3 .`	칸 쓰기(`mut` 로 받은 값)	읽기와 쓰기가 같은 철자
`enum shape do dot . circle r u32 . end`	여럿 중 하나 — 갈래는 값을 지닐 수 있다	갈래마다 마침표로 닫는다
`shape.circle 2` · `dot`	값을 지닌 갈래 만들기 · 값이 없는 갈래	갈래 이름이 곧 만드는 op
`match s do case circle r . … end`	갈래를 가르고 지닌 값을 묶기	모든 갈래를 덮어야 한다
`isa s circle`	그 갈래인가(`bool`)	값을 꺼내지 않고 묻기만 할 때
번호로 잇는 나무(`l u32` · `r u32`)	자기 자신을 품는 대신 슬라이스의 번호	크기가 정해지고 경계 검사를 받는다

표 10.1 — 묶음의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`struct` 는 `make` 로 모든 칸을 채워 만들고 `field` 로 읽고 쓴다. 붙임 점은 없다. `enum` 의 갈래는 값을 지닐 수 있고, `<타입>.<갈래>` 로 만들며 `match` 로 모든 갈래를 덮어 가른다. 갈래는 마침표로 닫는다. 자기 자신을 값으로 품는 타입은 크기가 무한하므로 거절되고, 나무 구조는 번호로 잇는다.

11 답을 담는 타입 — option 과 result

이 장의 차례

11.1 option — 값이 있거나 없거나 124
11.2 result 와 errors 절 125
11.3 try — 실패를 위로 넘긴다 127
11.4 두 채널 사이를 건넌다 127
11.5 실패를 말하는 세 길 128
11.6 패턴을 묶고 겹친다 129
11.7 흔한 실수 131
11.8 이 장의 문법 한눈에 135

먼저 알아야 할 것

7장 흐름 · guard 와 panic
10장 묶음 · 값을 지닌 enum 과 match

돌아보기

10장의 case rect w h . 는 무엇을 하는가? 그리고 match 가 갈래 하나를 빠뜨리면 어떻게 되는가?

> **답.** 갈래가 rect 이면 그 갈래가 지닌 두 값을 w 와 h 에 묶는다. 갈래를 빠뜨리면 E-MATCH-INEXHAUSTIVE 로 거절된다. 이 장의 option 과 result 는 언어가 미리 만들어 둔 두 갈래짜리 열거형처럼 행동한다 — 값이 있거나 없거나, 성공이거나 실패거나.

이 장의 필요성과 맥락

실패를 −1 이나 널 포인터 같은 특별한 값으로 알리는 관습은 "이 −1 은 오류인가 그냥 −1 인가" 를 소스만 보고 알 수 없게 만든다. 그리고 확인을 빠뜨려도 아무도 모른다. option 과 result 는 그 물음을 타입으로 옮긴다. 데이터를 다루는 제3부의 한가운데에 이 장이 있는 것은, 조회·변환·파싱처럼 데이터를 꺼내는 거의 모든 op 이 "없을 수 있음" 이나 "실패할 수 있음" 을 돌려주기 때문이다.

이 장이 끝나면

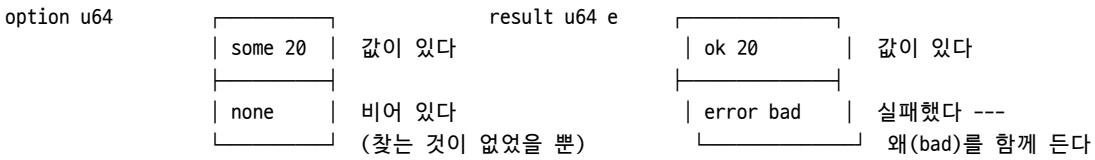
option 을 some·none 으로 만들고, 묻고 꺼내거나 value_or 로 대신할 값을 주거나 match 로 가르는 법을 익힌다. 확인 없이 꺼내면 멈춘다는 것을 알게 된다. result 와 errors 절이 짝을 이루고, try 가 실패를 위로 넘기며, else_none·else_error 꼬리가 둘 사이를 건넌다는 것도 보게 된다. 실패를 말하는 세 길 — result·option·panic — 을 가르는 기준을 세우고, or 패턴·겹친 패턴·번역 시점에 접히는 match 도 보게 된다.

이 장에서 답할 질문

1 value_or 의 기본값 자리에 비싼 계산을 적으면 매번 계산되는가?

11.1 option — 값이 있거나 없거나

option t 는 t 값이 있거나(some v) 없거나(none) 둘 중 하나다. 뒤에서 볼 result 와 함께, 값을 담는 **상자 두 종류**로 그려 두면 구별이 쉽다.



none 은 이상한 일이 아니라 «없었다» 는 정상적인 답이고, error 는 «하려던 일이 실패했다» 는 답이다. 어느 상자든 **열어 보기 전에는** 안의 값을 쓸 수 없다.

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
```

실행 결과

```
$ 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)
```

만드는 쪽인 `find` 는 `return none .` 과 `return some (mul k 10) .` 으로 값을 싼다. VM 은 결과를 `some 20` 과 `none` 으로 보여 준다. 받는 쪽은 세 가지로 쓸 수 있다.

- `find_or` — `value_or (find k) 99` 는 값이 있으면 그 값을, 없으면 99 를 준다.
- `find_asked` — `guard is_some r . else …` 로 먼저 묻고 `some_value r` 로 꺼낸다.
- `find_match` — `match` 로 `case some v .` 와 `case none .` 을 가른다. 두 갈래로 모든 경우가 덮인다.

`find_raw` 는 묻지 않고 바로 꺼낸다. 번역은 통과하지만 값이 없는 7 에서 실행이 멈춘다(`E-VM-NONE`). 꺼내는 연산은 **부분 연산**이다. 어느 길에서 값이 있는지는 저자가 아는 것이고 처리기가 언제나 알 수는 없으므로 번역이 막지 않는다. 대신 틀렸을 때 조용하지 않다 — 0 을 내주고 계속 가지 않는다.

> **문.** `value_or` **의 기본값 자리에 비싼 계산을 적으면 매번 계산되는가?**

> **답.** 아니다. 기본값은 **값이 없을 때만** 계산된다. `value_or (some 7) (div 1 0)` 은 7 이고 0 나누기는 일어나지 않는다. 그래서 기본값 자리에 실패할 수 있는 계산을 적어도 된다. 이 동작은 한때 반대였다가 명세에 맞게 고쳐졌다.

11.2 `result` 와 `errors` 절

`result t e` 는 성공한 값(`ok v`) 또는 오류(`error <갈래>`) 중 하나다. 오류 타입 `e` 는 보통 `enum` 이다. 그리고 `result` 를 돌려주는 op 은 **언제 어떤 오류를 내는지** `errors` 절에 적는다.

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
```

실행 결과

```
$ 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` 의 머리에는 오류가 둘 적혀 있다. `errors too_big gt a 200 .` 은 "`a` 가 200 보다 크면 `too_big` 을 낸다" 는 약속이다. 이 절은 **나가는 쪽의 계약**이다. 조건이 참인데 op 이 정상으로 돌아오거나, 적지 않은 오류를 돌려주면 계약 위반이다. 적지 않은 오류를 돌려주는 코드는 번역에서 거절된다.

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
```

실행 결과

```
$ lowentc --check undeclared.low
undeclared.low:13:0 E-ERR-UNDECLARED: op returns an error not in its errors clause
```

too_long 은 parse_error 의 갈래이지만 first_byte 의 errors 절에는 없다. 부르는 쪽은 머리만 보고 empty 만 다루면 된다고 믿을 것이다. 적은 것과 내는 것이 어긋나면 한쪽은 못 본 실패를, 다른 쪽은 오지 않을 실패를 다루게 된다.

11.3 try — 실패를 위로 넘긴다

실패를 확인하는 코드를 매번 손으로 쓰면 길어지고, 길어지면 빼먹는다. try 는 result 를 받아 성공이면 값을 꺼내고, 실패면 **그 오류를 그대로 돌려주며 op 을 떠난다.** halve.low 의 halve_plus_one 이 그 모양이다.

```
let v u8 be try halve a .
return ok (add v 1) .
```

halve_plus_one 이 자기 머리에도 errors 를 적은 것에 주목한다. try 로 오류를 넘기려면 자기도 그 오류를 돌려줄 수 있어야 하고, 그 사실을 계약에 적어야 한다. 실패가 조용히 사라지는 길이 없다. 실패를 위로 넘기지 않고 이 자리에서 다루려면 halve_or_zero 처럼 is_error 로 묻고 ok_value 로 꺼낸다.

> **흔한 오해. try 는 예외 처리의 try 다**
>
> Java 나 C++ 의 try 는 블록 안 어디서든 던져진 예외를 잡는 자리다. Lowent 의 try 는 **식 하나**에 붙어서 그 식의 실패를 **위로 넘기는** 표시이고, Rust 의 ? 에 가깝다. 던지고 잡는 제어 흐름은 없다. 실패는 언제나 값으로 돌아오고, 어느 식에서 넘어갈 수 있는지가 소스에 적혀 있다.

11.4 두 채널 사이를 건넌다

부르는 op 과 불리는 op 의 채널이 다를 때가 있다. try 뒤에 꼬리를 붙이면 담는 그릇을 바꾼다.

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
```

실행 결과

```
$ 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
```

모양	방향	무엇을 잃거나 얻나
`try <식> else_none`	`result` → `option`	오류를 **버린다**. 왜 실패했는지 더는 말하지 않는다
`try <식> else_error <갈래>`	`option` → `result`	없음에 **이름을 붙인다**

표 11.1 — `try` 의 꼬리로 채널을 바꾼다

꼬리를 붙인 `try` 는 타입도 바꾼다. `try (halve a) else_none` 의 타입은 `option u8` 이지 `u8` 이 아니다. 그래서 `maybe_half` 는 그것을 그대로 돌려주고, `let v u8 be try … else_error …` 처럼 값 타입에 담으려 하면 거절된다.

`else_none` 은 정보를 버리는 선택이다. 편해서 습관이 되기 쉬운데, 그 순간부터 호출자는 "왜" 를 물을 수 없다. 버릴 만한 자리에서만 버린다.

11.5 실패를 말하는 세 길

무엇	무엇을 말하나	부르는 쪽이 하는 일
`result t e`	고칠 수 있는 실패	어느 쪽인지 묻고 다룬다. 안 다루면 위로 넘긴다
`option t`	값이 없음	있는지 묻고 꺼내거나, 대신 쓸 값을 준다
`panic` · **계약 위반**	약속이 깨졌다	다룰 수 없다. 프로그램이 멈춘다

표 11.2 — 실패를 말하는 세 길

셋을 가르는 물음은 "부르는 쪽이 무엇을 할 수 있는가" 다. 파일이 없으면 다른 파일을 열어 볼 수 있으니 `result` 다. 찾는 것이 목록에 없으면 그냥 없는 것이니 `option` 이다. 부르는 쪽이 계약을 어겼다면 이미 약속이 깨진 것이라 고칠 수 있는 일이 아니고, 그래서 멈춘다. `panic` 은 되돌아 풀리지 않는다. 중간에 잡아 이어 가는 길은 없다. 이 기준으로 op 의 실패를 설계하는 법은 17장에서 다시 다룬다.

11.6 패턴을 묶고 겹친다

option·result·enum 을 모두 보았으니 match 의 패턴을 더 넓게 쓸 수 있다.

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 어느 하나라도 맞으면 그 갈래 --- red·green·blue 를 모두 덮으므로 _ 가 필요 없다
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 or 로 묶은 갈래는 같은 이름을 묶어야 한다
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 result 안의 option 을 한 번에 가른다
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 가르는 값이 번역 시점 상수면 맞는 갈래 하나로 접힌다
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 두 범위가 u8 전체를 빈틈 없이 덮으므로 _ 가 필요 없다
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
```

실행 결과

```
$ 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 패턴.** `case red or green .` 은 어느 하나라도 맞으면 그 갈래다. 갈래마다 망라에 기여하므로 `blue` 까지 다루면 `_` 가 필요 없다. 값을 지닌 갈래를 or 로 묶을 때는 **모든 가지가 같은 이름을 묶어야** 한다 — `combine` 은 `plus` 든 `times` 든 `l·r` 을 꺼내 쓴다. 이름이 어긋나면 `E-MATCH-ORBIND` 다.
- **겹친 패턴.** `case ok (some x) .` 는 `result` 안의 `option` 을 한 번에 가른다. 겹친 패턴은 단락 평가라서 `ok` 가 아니면 안쪽을 아예 보지 않는다. 그래서 `error` 에서 값을 잘못 꺼내 멈추는 일이 없다.
- **번역 시점에 접힌다.** 가르는 값이 번역 시점 상수 — 리터럴, `comptime <식>`, `config <이름>` — 이면 `match` 는 맞는 갈래 하나로 접혀 실행 중 비교가 없다. 죽은 갈래도 타입 검사는 받는다. C 의 `#ifdef` 와 다른 점이다(31장).
- **범위가 타입을 덮는다.** `half` 의 두 범위는 `u8` 의 0 … 255 를 빈틈 없이 덮으므로 `_` 가 없어도 망라다. 빈틈이 있으면 `E-MATCH-INEXHAUSTIVE`, 범위가 겹치면 `E-MATCH-REDUNDANT` 다.

  `_` 뒤에 갈래를 두면 거절된다.

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
```

실행 결과

```
$ 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)
```

`_` 가 이미 모두 받았으므로 뒤의 갈래는 영영 돌지 않는다. 죽은 갈래는 경고가 아니라 오류다 — 읽을 때 각 경우가 한 번씩이 아닌 `match` 는 결함을 숨긴다.

11.7 흔한 실수

반례. `option` 을 수처럼 계산에 넣는다

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 ✘ `option u64` 는 수가 아니다 --- 있는지 먼저 묻고 꺼내야 더할 수 있다
  return add (find k) 1 .
end
```

실행 결과

```
$ 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` 가 돌려주는 것은 `u64` 가 아니라 "`u64` 가 있을 수도 없을 수도 있는 상자" 다. 상자에 1 을 더할 수는 없다. 다른 언어라면 없음(null)이 계산 속으로 흘러 들어가 한참 뒤에 터지지만, Lowent 는 여기서 `E-TYPE-RETURN` 으로 멈춰 세운다. 고치는 길은 셋이다. `value_or (find k) 0` 으로 대신할 값을 주거나, `is_some` 으로 묻고 `some_value` 로 꺼내거나, `match` 로 가른다. 어느 것을 고를지는 "없을 때 무엇을 해야 하는가" 가 정한다.

반례. `option` 을 돌려주는 op 에서 `some` 을 빠뜨린다

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 ✘ 값을 `some` 으로 싸지 않았다
  return mul k 10 .
end
```

실행 결과

```
$ 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`
```

머리에 `output option u64` 라고 적었으면 돌려주는 값도 상자여야 한다. `none` 은 상자이지만 `mul k 10` 은 맨 수라서 `E-TYPE-RETURN` 이다. 몇몇 언어는 값을 알아서 감싸 주지만 Lowent 는 감싸지 않는다. `return some (mul k 10) .` 처럼 "있다" 를 적어야 읽는 사람이 두 갈래를 모두 본다.

반례. `result` 를 돌려주지 않는 op 에서 `try` 를 쓴다

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 ✘ `try` 를 썼는데 이 op 은 `result` 를 돌려주지 않고 `errors` 절도 없다
fn plus input a u8 . output u8 .
do
  let v u8 be try halve a .
  return add v 1 .
end
```

실행 결과

```
$ 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` 는 실패를 **위로 넘긴다**. 그러려면 이 op 도 실패를 돌려줄 수 있어야 한다(§ 6.5.8(2)). 그런데 `plus` 는 `u8` 만 돌려주고 `errors` 절도 없어서 `E-TRY-NORESULT` 로 거절된다. 2026-09-16 까지는 통과했고, `plus 250` 은 `u8` 자리에 `err too_big` 을 냈으며 네이티브는 C 컴파일 단계에서 지어지지도 않았다. 꼬리를 붙여 채널을 바꾸거나(`else_none`·`else_error`), `is_ok` 처럼 여기서 다루면 `result` 를 돌려주지 않아도 된다. 실패를 위로 넘길 생각이라면 머리를 `result` 와 `errors` 로 맞춘다.

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 실패를 위로 넘기는 op 은 자기도 result 를 돌려주고, 넘길 수 있는 오류를 errors 에 적
는다
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
```

실행 결과

```
$ lowentc --run plus trynoresult_fixed.low 10
plus(10) = ok 6
$ lowentc --run plus trynoresult_fixed.low 250
plus(250) = err too_big
```

겹친 패턴도 망라에 센다. 바깥 꼬리표는 **안쪽이 스스로 빠짐없을 때** 덮인 것으로 세어진다.

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 겹친 패턴 셋이 모든 경우다 --- `_` 를 따로 두지 않는다
  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
```

실행 결과

```
$ lowentc --run via_match nestedwild.low
via_match() = 42
```

`ok (some x)` 와 `ok none` 이 함께 `ok` 를 덮고 `error e` 가 나머지를 덮으므로 `_` 가 필요 없다. 하나라도 빠지면 `E-MATCH-INEXHAUSTIVE` 로 거절된다 — 쓸모없는 `_` 를 두는 것보다 낫다. `_` 는 나중에 갈래가 늘어도 아무 말을 하지 않는다.

흔한 오해. `value_or` 를 쓰면 없는 경우도 알 수 있다

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 0 번 칸에는 진짜 0 이 있고, 7 번 칸은 없다 --- 그런데 둘 다 0 으로 보인다
fn find_or_zero input k u8 . output u8 .
do
  return value_or (find k) 0 .
end

rem 있는지를 알고 싶으면 대신할 값으로 덮기 전에 묻는다
fn found input k u8 . output bool .
do
  return is_some (find k) .
end
```

실행 결과

```
$ 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` 는 없음을 **대신할 값으로 덮는다**. 덮는 값이 진짜 값과 겹치면 둘을 가를 수 없다. 위에서 0 번 칸의 진짜 0 과 7 번 칸의 없음이 똑같이 0 이다. “없으면 0 으로 쳐도 된다” 가 맞는 자리에서만 `value_or` 를 쓰고, 있는지가 중요하면 덮기 전에 `is_some` 이나 `match` 로 묻는다.

11.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`output option u8 .`	값이 있거나 없다	없음(null)을 타입에 드러낸다
`some v • none`	있다 · 없다	“있다” 도 적어야 두 갈래가 모두 보인다
`output result u8 e .`	값이거나 오류(`e` 의 갈래)	실패도 값으로 돌려준다 — 예외가 없다
`ok v • error too_big`	성공 · 실패	어느 갈래인지 소스에 적힌다
`errors too_big <조건> .`	어떤 오류가 언제 나는지 약속	계약에 적어 부르는 쪽이 대비한다
`is_some r • some_value r`	있는지 묻기 · 꺼내기	꺼내기는 부분 연산 — 묻고 꺼낸다
`is_error r • ok_value r`	실패인지 묻기 · 성공 값 꺼내기	같은 이유
`value_or r 99`	없으면 대신할 값	한 줄로 끝나지만 없음을 덮는다
`try <식>`	실패면 그 오류를 돌려주며 떠난다	확인 코드를 빼먹지 않게 — Rust 의 `?`
`try <식> else_none • else_error e`	`result` → `option` · `option` → `result`	채널을 바꿀 때 무엇을 잃는지 드러낸다
`case ok (some x) . • case a or b .`	겹친 패턴 · 여러 갈래 묶기	한 번에 가르되 모든 경우를 덮는다

표 11.3 — 답을 담는 타입의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

> **복습 정리**
>
> `option` 은 `some`·`none`, `result` 는 `ok`·`error` 로 만든다. 받는 쪽은 묻고 꺼내거나(`is_some`·`some_value` · `is_error`·`ok_value`), `value_or` 로 대신할 값을 주거나, `match` 로 가른다. 꺼내기는 부분 연산이라 없는 쪽을 꺼내면 멈춘다. `result` 를 돌려주는 op 은 `errors` 절로 오류를 약속하고, `try` 는 실패를 위로 넘기며, `else_none`·`else_error` 꼬리는 채널과 타입을 바꾼다. 패턴은 `or` 로 묶고(같은 이름을 묶는다) 겹쳐 쓸 수 있으며, 상수를 가르는 `match` 는 번역에서 접힌다.

12 빌리기 — `ref` 와 `mut_ref`

이 장의 차례

12.1 두 가지 빌림 136
12.2 쓰기 권한은 한 방향으로만 좁아진다 138
12.3 읽기 여럿 또는 쓰기 하나 138
12.4 빌림은 빌려준 값보다 오래 살 수 없다 140
12.5 `mut ref slice` 는 없다 140
12.6 흔한 실수 141
12.7 이 장의 문법 한눈에 144

먼저 알아야 할 것

9장 줄 · `mut slice` 여야 원소에 쓸 수 있다
10장 묶음 · `field` 는 읽는 자리이자 쓰는 자리다

돌아보기

9장에서 `mut` 가 없는 슬라이스의 원소에 쓰면 무엇으로 거절되었는가? 그리고 `let` 으로 지은 이름의 불변과 슬라이스 원소의 불변은 어떻게 다른가?

> **답.** `E-TYPE-MUT` 로 거절되었다. `let` 은 **이름에 다른 값을 다시 넣는 일**을 막고, 원소에 쓸 수 있는지는 슬라이스의 타입(`slice` 인가 `mut slice` 인가)이 정한다. 이 장은 슬라이스가 아닌 값 하나를 빌리는 참조와, 빌림이 겹칠 때의 규칙을 다룬다.

이 장의 필요성과 맥락

큰 값을 넘길 때마다 베끼면 느리고, 포인터로 넘기면 누가 언제 그 값을 바꾸는지 알 수 없다. Lowent는 그 사이에 **빌리기**를 둔다. 읽기로 빌리는 쪽은 여럿일 수 있고, 쓰기로 빌리는 쪽은 혼자여야 한다. 이 규칙이 쓰레기 수거기 없이 메모리를 안전하게 하는 바탕이며, 동시성(제7부)에서 데이터 경합이 없다는 증명도 이 규칙 위에 선다. 메모리(제5부)로 가기 전에 빌리기의 모양부터 익힌다.

이 장이 끝나면

`ref t` 와 `mut_ref t` 로 값을 빌리고 `deref` 로 읽는 법을 익힌다. 읽기 빌림으로는 쓸 수 없고, 쓰기 권한은 한 방향으로만 좁아진다는 것을 알게 된다. 한 값에 대한 빌림이 "읽기 여럿 또는 쓰기 하나" 여야 한다는 배타 규칙과, 지역을 가리키는 참조가 op 밖으로 나갈 수 없다는 규칙을 보게 된다. `mut ref slice` 가 왜 거절되는지도 이해하게 된다.

이 장에서 답할 질문

1 널 참조는 어떻게 만드는가?

12.1 두 가지 빌림

참조는 다른 곳에 있는 값을 가리킨다. 두 갈래뿐이고, 이름이 곧 권한이다.

- `ref t` — 읽기 빌림. 여럿이 동시에 가질 수 있다. 이것으로는 값을 바꿀 수 없다.
- `mut_ref t` — 쓰기 빌림. 한 값에 대해 한 번에 하나만 있을 수 있고, 그동안 읽기 빌림도 있을 수 없다.

도서관 책에 빗대면 쉽다. 여럿이 한꺼번에 **열람**(읽기)할 수는 있지만, 책에 **고쳐 적으려면**(쓰기) 혼자 빌려 가야 하고, 그동안은 아무도 열람하지 못한다. 고치는 동안 누가 읽으면 반쯤 고친 내용을 보게 되기 때문이다.

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
```

실행 결과

```
$ lowentc --run use_sum borrow.low 7
use_sum(7) = 32
$ lowentc --run use_bump borrow.low 7
use_bump(7) = 16
```

- `total` 은 네 칸짜리 `big` 을 베끼지 않고 `ref big` 으로 빌려 읽는다. `field v a` 는 참조를 거쳐 칸을 읽는다.

- `use_sum` 은 같은 값을 `ref v` 로 두 번 빌려 넘긴다. 읽기 빌림은 여럿이어도 된다.
- `bump` 는 `mut_ref u64` 를 받아 `set p …` 로 호출자의 값을 바꾼다. 참조가 가리키는 값은 `deref p` 로 읽는다.
- `use_bump` 는 `var n` 을 `mut_ref n` 으로 빌려준다. `bump` 가 돌아온 뒤 `n` 은 8 이다.

빌리는 쪽이 `ref x` 나 `mut_ref n` 을 **부르는 자리에 적는다**는 점이 중요하다. 호출을 읽는 사람은 그 자리만 보고도 이 호출이 `n` 을 바꿀 수 있다는 것을 안다.

> **문. 널 참조는 어떻게 만드는가?**
>
> **답.** 만들 수 없다. 참조는 언제나 살아 있는 값을 가리킨다. “가리키는 것이 없음” 을 나타내야 하면 `option` 을 쓴다(11장). 그러면 꺼내기 전에 확인하는 코드가 소스에 드러나고, 확인하지 않고 꺼내면 멈춘다.

12.2 쓰기 권한은 한 방향으로만 좁아진다

읽기 빌림으로 쓰려 하면 거절된다.

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
```

실행 결과

```
$ lowentc --check ref_write.low
ref_write.low:6:0 E-TYPE-REF: write through a shared ref (declare mut_ref)
```

반대 방향도 막힌다. 읽기로만 받은 값을 `mut`·`mut_ref`·`owned` 를 받는 자리에 넘기면 `E-TYPE-ARGMUT` 으로 거절된다. 쓸 수 있는 것을 읽기로 건네는 것은 되지만, 읽기만 되는 것을 쓸 수 있는 자리로 건넬 수는 없다. 어떤 값을 읽기로 받은 쪽은 **자기가 보는 동안 그 값이 바뀌지 않는다**고 믿을 수 있어야 하고, 그 믿음은 아무도 몰래 쓰기 권한을 얻지 못할 때에만 선다.

12.3 읽기 여럿 또는 쓰기 하나

같은 값에 대한 빌림이 겹칠 때가 문제다. 쓰기 빌림 둘이 겹치면 거절된다.

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
```

실행 결과

```
$ lowentc --check excl.low
13:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
```

`set_both` 의 본문은 `a` 와 `b` 가 서로 다른 값이라고 믿는다. 같은 `n` 을 두 번 빌려주면 그 믿음이 깨지고, 남는 값이 1 인지 2 인지는 쓰는 차례에 달린다. 읽기와 쓰기가 겹쳐도 거절된다.

examples/ch12/stale.low

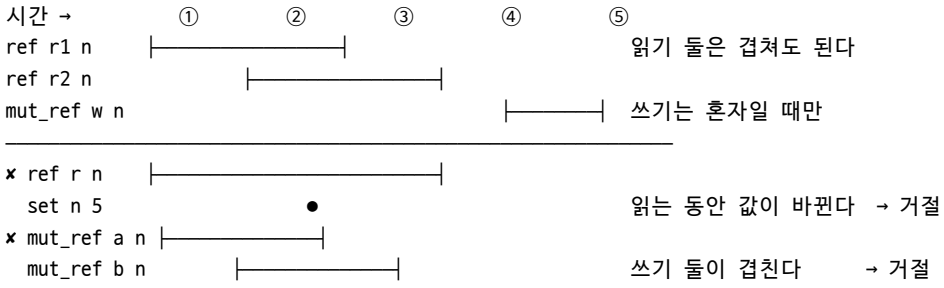
```
module stale .
rem expect: E-EXCL

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
```

실행 결과

```
$ lowentc --check stale.low
8:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
```

`r` 로 `n` 을 읽기로 빌린 채 `n` 에 5 를 쓰면, `r` 을 읽는 쪽은 자기가 본 값이 언제 바뀌었는지 알 수 없다.

같은 값 `n` 에 대한 빌림을 시간 순으로 늘어놓으면 규칙이 한눈에 보인다.



이 두 모양 — 다른 흐름이 동시에 고치는 것, 읽는 중에 값이 바뀌는 것 — 은 소스를 읽어서는 찾기 어려운 결함이라 규칙으로 없앤다.

> **흔한 오해. 배타 규칙은 멀티스레드 프로그램에만 필요하다**
>
> 위의 두 예제에는 스레드가 하나도 없다. 한 흐름 안에서도 두 이름이 같은 값을 가리키면 한쪽의 쓰기가 다른 쪽의 믿음을 깬다. C 의 `memcpy` 가 겹치는 버퍼에서 미정의 동작인 것도 같은 이유다. 다만 이

> 규칙이 한 흐름 안에서 지켜지면 여러 흐름으로 나누었을 때 데이터 경합이 없다는 것까지 따라 나오고, 그것이 제7부의 증명이 기대는 자리다(45장).

12.4 빌림은 빌려준 값보다 오래 살 수 없다

op 안의 지역을 가리키는 참조를 op 밖으로 돌려주면 거절된다.

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
```

실행 결과

```
$ lowentc --check escape.low
7:0 E-ESCAPE: reference to a local escapes the op (dangling)
```

만약 이 코드가 번역된다면, 돌려받은 참조는 이미 사라진 값을 가리킨다. 그 참조를 읽으면 무슨 값이 나올지 아무도 모른다. Lowent 는 이것을 실행해 보고 아는 것이 아니라 **번역할 때** 안다. 빌림은 그것을 연 블록 밖으로 나갈 수 없고(`E-BORROW-ESCAPE`), 빌려준 값이 옮겨졌는데 빌림이 남아 있어도 거절된다(`E-EXCL-MOVED`).

값을 op 밖으로 내보내야 한다면 참조가 아니라 **값**을 돌려주거나, 호출자가 넘긴 저장소에 담는다. 호출자보다 오래 사는 저장소가 필요하면 영역을 쓴다(18장).

12.5 `mut ref slice` 는 없다

슬라이스에는 `mut ref` 를 붙이지 않는다.

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
```

실행 결과

```
$ 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)
```

`mut slice` 를 넘기면 원소는 이미 쓸 수 있다. 슬라이스 값 `{시작, 길이}` 는 베껴 가지만 가리키는 바이트는 같기 때문이다. 그러니 `mut ref slice` 가 더 주는 능력은 하나뿐이다 — **호출자 쪽 슬라이스의 길이와 시작을 몰래 바꾸는 것.** 그러면 호출자의 반복문은 자기 코드 어디에도 대입이 없는데 길이가 달라진다. 새 슬라이스는 돌려주면 된다.

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
```

실행 결과

```
$ 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` 는 앞쪽 `n` 개를 가리키는 새 슬라이스를 돌려준다. 길이가 달라졌다는 사실이 반환값과 `let front` 에 드러난다.

12.6 흔한 실수

반례. 부르는 자리에 `mut_ref` 를 빠뜨린다

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 x 부르는 자리에 `mut_ref` 를 빠뜨렸다 --- 참조가 아니라 수 7 이 넘어간다
  let r u64 be bump n .
  return add n r .
end
```

실행 결과

```
$ 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` 는 "수를 가리키는 참조" 를 받기로 했는데 `bump n` 은 수 7 을 그대로 넘긴다. C++ 의 참조 매개변수는 부르는 자리에 아무 표시가 없어도 되지만, Lowent 는 호출만 보고도 무엇이 바뀔지 알게 하려고 `mut_ref n` 을 적게 한다. 이 판의 도구는 이 빠뜨림을 번역에서 거절하지 않고, 실행하면 `deref` 가 참조가 아닌 것을 만나 멈춘다(개발 저장소에 결함으로 적어 두었다). 멈춤의 메시지가 "deref needs a reference" 이면 부르는 자리에서 `mut_ref` 나 `ref` 를 찾는다.

반례. 참조를 `deref` 없이 수처럼 쓴다

examples/ch12/mistake_noderef.low

```
module mistake_noderef .
rem expect: E-TYPE-REFVAL

fn twice input p ref u64 . output u64 .
do
  rem x `p` 는 수가 아니라 수를 가리키는 참조다 --- `deref p` 로 읽어야 한다
  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
```

실행 결과

```
$ 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
```

참조는 값이 **있는 곳**이지 값이 아니다. 읽을 때마다 `deref p` 라고 적어서 "여기서 빌린 값을 읽는다" 를 드러낸다. C 에서 `*p` 를 빠뜨리면 주소끼리 더하는 엉뚱한 계산이 되지만, Lowent 는 주소 계산을 허용하지 않으므로 잘못된 값이 조용히 나오지는 않는다. 다만 이 판의 도구는 번역에서 알리지 않고 실행 중에 멈춘다(결함으로 적어 두었다). 고친 줄은 `add (deref p) (deref p)` 다.

반례. `let` 으로 지은 이름을 쓰기로 빌려준다

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 ✘ `let` 으로 지은 이름인데 쓰기로 빌려준다
  let n u64 be start .
  bump (mut_ref n) .
  return n .
end
```

실행 결과

```
$ 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` 은 "이 이름의 값은 바뀌지 않는다" 는 약속이다. 그런 이름을 `mut_ref` 로 빌려주면 약속이 깨지므로 `E-TYPE-ARGMUT` 로 거절한다. 빌림은 부르는 자리에서 보이지 않게 값을 바꾸기 때문에, 한 번 확인하면 된다는 `let` 의 값이 사라진다. 바뀌어야 하는 값은 처음부터 `var` 로 짓는다. 그러면 읽는 사람이 "이 값은 어딘가에서 바뀐다" 를 선언에서 안다.

흔한 오해. `input` 으로 넘긴 변수는 불린 op 이 바꿀 수 있다

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` 는 호출자가 넘긴 값의 사본이다 --- 여기서 무엇을 해도 호출자의 변수에 닿지 않
는다
  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` 은 그대로 7 이다 --- 바꾸게 하려면 `mut_ref n` 으로 빌려준다
  return n .
end
```

실행 결과

```
$ lowentc --run use_inc copyparam.low 7
use_inc(7) = 7
```

맨 값 매개변수(`input x u64`)는 호출자의 값을 **베껴서** 받는다. `inc` 가 안에서 무엇을 하든 `use_inc` 의 `n` 은 7 그대로다. 어떤 언어는 큰 값을 몰래 참조로 넘겨 이 경계가 흐리지만, Lowent 에서 호출자의 값을 바꾸는 길은 `mut_ref` 하나이고 그 표시는 부르는 자리에 남는다.

흔한 오해. 한 op 안에서는 같은 값을 `mut_ref` 로 한 번만 빌릴 수 있다

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 첫 빌림은 이 호출이 끝날 때 돌아온다 --- 그래서 다음 줄에서 또 빌릴 수 있다
  bump (mut_ref n) .
  bump (mut_ref n) .
  return n .
end
```

실행 결과

```
$ lowentc --run use_twice seqborrow.low 7
use_twice(7) = 9
```

배타 규칙이 막는 것은 **동시에 겹친** 빌림이다. 호출 하나에 넘긴 `mut_ref n` 은 그 호출이 끝나면 돌아오므로 다음 줄에서 또 빌릴 수 있다. 거절되는 것은 `set_both (mut_ref n) (mut_ref n)` 처럼 한 호출 안에 두 빌림이 함께 살아 있는 모양이다.

12.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`input v ref big .`	읽기로 빌려 받는다	큰 값을 베끼지 않고, 바꾸지 않는다고 약속한다

`input p mut_ref u64 .`	쓰기로 빌려 받는다	호출자의 값을 바꿀 수 있는 유일한 길
`total (ref v) · bump (mut_ref n)`	부르는 자리에서 빌려준다	호출만 보고도 무엇이 바뀔지 안다
`deref p`	참조가 가리키는 값을 읽는다	값과 값이 있는 곳을 섞지 않는다
`set p <값> .`	참조가 가리키는 곳에 쓴다 (`mut_ref`)	읽기 빌림으로는 `E-TYPE-REF`
`field v a`	참조를 거쳐 칸을 읽는다	구조체에 대한 참조도 같은 철자
같은 값에 `mut_ref` 둘 · `ref` 와 쓰기	겹친 빌림 — 거절(`E-EXCL`)	읽기 여럿 또는 쓰기 하나
같은 저장소를 쓰기 자리와 읽기 자리에	피호출자가 `inplace <쓰기> <읽기> .` 로 허락했고 **같은 구간**일 때만 — 아니면 `E-EXCL-INPLACE`	원소마다 읽고 쓰는 몸은 같은 구간에서만 옳다
`inplace o a` **를 적은 op 의 몸**	`a` 를 다 읽은 뒤에 `o` 에 쓰거나, 한 첨자로 원소마다 읽고 쓴다 — 아니면 `E-INPLACE-UNPROVEN`	선언이 참인지 처리기가 몸에서 본다
`invalidates <입력>` **을 밝힌 op 을 부른 뒤 그 저장소의 옛 뷰를 쓴다**	거절(`E-VIEW-INVALIDATED`) — 뷰를 다시 받는다	반환·성장·되감기 뒤의 뷰는 남의 자리를 본다
`return ref here .`	지역을 가리키는 참조를 돌려준다 — 거절	사라진 값을 가리키지 않게
`mut ref slice`	없다 — 거절(`E-MREF-SLICE`)	길이를 몰래 바꾸지 않게 — 새 슬라이스는 돌려준다

표 12.1 — 빌리기의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`ref t` 는 읽기, `mut_ref t` 는 쓰기 빌림이고, 빌리는 쪽이 부르는 자리에 `ref x`·`mut_ref n` 을 적는다. 참조가 가리키는 값은 `deref` 로 읽는다. 쓰기 권한은 한 방향으로만 좁아진다. 한 값에 대한 빌림은 읽기 여럿 또는 쓰기 하나이며, 빌림은 빌려준 값보다 오래 살 수 없다. `mut ref slice` 는 거절되고, 새 슬라이스는 돌려준다.

13 이름 붙인 타입 — type • newtype • range • cast

이 장의 차례

13.1 type 은 별칭, newtype 은 새 타입 147
13.2 range — 매개변수의 생김새가 된 계약 149
13.3 cast — 값이 변할 수 있는 자리 150
13.4 bits — 1 부터 64 비트까지 151
13.5 바이트 배치를 못 박는다 151
13.6 배치의 두 방향과 실패하는 뷰 152
13.7 bitset — 작은 수의 집합 153
13.8 흩어진 조각을 한 눈으로 — segments 155
13.9 타입 낱말의 세 갈래 155
13.10 흔한 실수 157
13.11 이 장의 문법 한눈에 159

먼저 알아야 할 것

4장 수 · 넓히기는 자동, 좁히기는 적는다
10장 묶음 · struct 와 field

돌아보기

4장에서 usize 와 u64 는 폭이 같은 기계에서도 서로 자동으로 바뀌지 않는다고 했다. 왜 그렇게 정했는가?

> **답.** 주소를 세는 수와 그냥 수를 섞는 자리를 소스에 드러내기 위해서다. 표현이 같아도 뜻이 다르면 다른 타입으로 두고, 건너는 자리를 적게 한다. 이 장은 같은 원칙을 사용자가 직접 쓰는 방법 — newtype — 과, 타입에 범위와 폭과 배치를 적는 방법을 다룬다.

이 장의 필요성과 맥락

사용자 번호와 주문 번호가 둘 다 u64 이면 바꿔 넣어도 컴파일러는 모른다. 퍼센트를 받는 매개변수에 300 을 넘겨도 모른다. 이런 결함은 타입이 **표현**만 말하고 **뜻**을 말하지 않아서 생긴다. 제3부의 마지막 장으로, 앞에서 배운 타입들에 이름과 범위와 배치를 붙여 뜻을 싣는 법을 모은다. 계약(제4부)으로 넘어가기 직전에 두는 것은 range 가 계약과 같은 자격으로 쓰이기 때문이다.

이 장이 끝나면

type 은 같은 타입의 다른 이름이고 newtype 은 같은 표현의 다른 타입이라는 것, 둘 사이를 cast 로 건넌다는 것을 알게 된다. 매개변수 자리의 range lo hi 가 부르는 쪽과 본문에 무엇을 주는지, cast 가 widen·narrow 와 어떻게 다른지를 익힌다. 1 부터 64 비트까지의 bits 정수와, 바이트 배치를 못 박는 layout packed 와 view 도 보게 된다.

이 장에서 답할 질문

1 newtype 을 쓰면 실행 비용이 드는가?

13.1 type 은 별칭, newtype 은 새 타입

`type <이름> <타입> .` 은 기존 타입에 다른 이름을 준다. 두 이름은 같은 타입이고 서로 바꿔 쓸 수 있다.

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
```

실행 결과

```
$ lowentc --run area aliases.low 3 4
area(3, 4) = 12
$ lowentc --run half_percent aliases.low 90
half_percent(90) = 45
```

`meters` 는 `u64` 의 다른 이름일 뿐이다. 그래서 `mul w h` 의 결과를 `u64` 로 돌려줄 수 있다. 별칭은 긴 타입을 짧게 부르거나(`type bytes slice u8 .`), 뜻을 소스에 남길 때 쓴다. `pct` 는 범위가 붙은 타입의 별칭이고, 이 이름을 받는 모든 자리가 그 범위를 물려받는다(아래에서 다룬다).

타입 선언에는 `be` 를 끼우지 않는다.

examples/ch13/type_be.low

```
module type_be .
rem expect: E-TYPE-DECL

type pct be u8 .
```

실행 결과

```
$ 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` 는 `let`·`var` 가 **값**을 묶는 낱말이다. 여기서 묶는 것은 타입이다. 한 뜻에 한 표기다.

`newtype <이름> <타입> .` 은 기존 타입과 같은 표현을 갖되 **다른 타입**을 만든다.

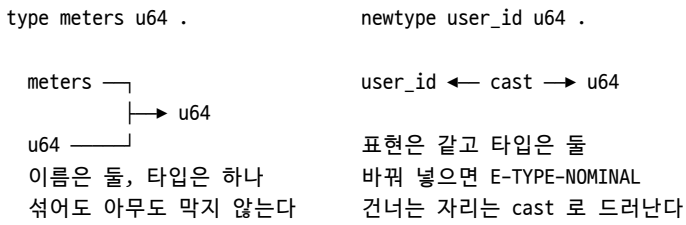
examples/ch13/ids.low

```
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
```

실행 결과

```
$ 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` 와 `order_id` 는 둘 다 64 비트 수로 표현되지만 서로 바꿔 넣을 수 없다. 이런 실수는 테스트로 찾기 어렵다 — 번호가 우연히 겹치면 엉뚱한 주문을 불러오고도 조용하다. `newtype` 으로 갈라 두면 번역이 잡는다.



일부러 건너야 할 때는 `cast` 로 적는다.

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
```

실행 결과

```
$ lowentc --run roundtrip ids_ok.low 42
roundtrip(42) = 42
```

`user_of` 와 `raw_of` 처럼 건너는 op 을 한두 개만 두면, 날것의 수가 `user_id` 로 들어오는 자리가 코드에서 그 둘로 모인다. 그 자리에서만 검사하면 된다.

> **문. `newtype` 을 쓰면 실행 비용이 드는가?**
>
> **답.** 들지 않는다. 표현이 같으므로 네이티브 코드에서 `user_id` 는 그냥 64 비트 정수다. 구별은 번역할 때만 있고, `cast user_id n` 은 값을 바꾸지 않는다. 비용 없는 구별이라서 적극적으로 쓸 만하다.

13.2 range — 매개변수의 생김새가 된 계약

매개변수의 타입 자리에 `range <아래> <위>` 를 적으면 그 매개변수는 두 끝을 포함한 그 사이의 값만 받는다.

examples/ch13/ranges.low

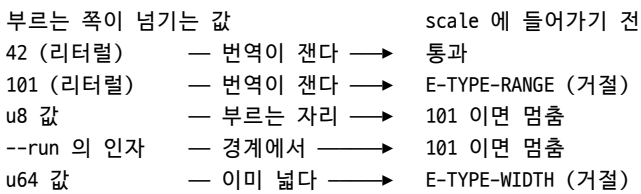
```
module ranges .
rem run: scale 100
rem trap: scale 101

fn scale input a range 0 100 . output u8 .
do
  return narrow u8 (mul a 2) .
end
```

실행 결과

```
$ 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)
```

`scale` 의 본문은 `a` 가 100 이하라는 사실을 쓴다. 그래서 `mul a 2` 는 200 을 넘지 않고, `narrow u8` 은 들어가지 않을 수 없으며, 두 자리의 넘침 검사가 모두 지워진다. 범위를 지키는 책임은 부르는 쪽에 있다. 넘기는 값의 타입이 이미 범위보다 넓으면(`u64` 따위) 번역이 거절되고, `u8` 처럼 들어갈 수 있는 타입의 값은 부르는 자리에서 범위를 확인한다. 프로그램 **바깥**에서 들어온 값(여기서는 `--run` 의 인자)도 경계에서 확인되고, 101 은 거기서 멈춘다. 명세는 부르는 쪽이 범위를 **증명하지 못하면** 번역을 거절한다고 적고 있으나, 이 판의 컴파일러는 증명되지 않은 `u8` 값을 실행 중 확인으로 넘긴다.

범위를 누가 어디서 재는지 한 장으로 보면 이렇다(`101` 리터럴은 이 판에서 실측했다).



같은 일을 `requires le a 100 .` 으로도 적을 수 있다. 차이는 **어디에 적히는가**다. `range` 는 매개변수의 생김새가 되어 부르는 쪽이 서명만 보고도 알고, 별칭(`type pct range 0 100 .`)에 실어 여러 op 에 물려줄 수 있다.

13.3 cast — 값이 변할 수 있는 자리

`widen` 은 값이 변하지 않는 자리에만 쓴다. 부호를 바꾸거나 갈래를 넘는(부동소수 ↔ 정수) 변환은 `cast <타입> <값>` 으로 적는다.

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
```

실행 결과

```
$ 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)
```

- 부동소수를 정수로 바꾸면 0 쪽으로 버린다. 7.9 는 7, −2.9 는 −2 다. 버려지는 소수부 때문에 멈추지는 않는다.
- 정수부나 값이 목표 타입에 들어가지 않으면 멈춘다. 30억은 `i32` 에 들어가지 않는다.
- `bool` 은 `cast` 의 대상도 원천도 될 수 없다. 참거짓과 수를 오가려면 `if` 로 적는다.

`cast` 는 "아무 일이나 해도 된다" 는 표시가 아니다. **여기서 값이 변할 수 있음을 내가 안다**는 표시다. 값을 잃는 변환이 암묵적으로 일어나지 않는다는 원칙(4장)을 어기지 않는 것은, 저자가 그 낱말을 적었기 때문이다.

> **흔한 오해. `cast` 는 C 의 캐스트처럼 비트를 그대로 옮긴다**
>
> C 의 `(int32_t)x` 는 들어가지 않는 값을 구현에 따라 자르거나 감는다. Lowent 의 `cast` 는 값을 옮기고, 들어가지 않으면 멈춘다. 비트를 그대로 두고 읽는 법만 바꾸는 일은 따로 `bit_cast` 가 맡는다 (20장). 두 일을 한 낱말에 싣지 않는다.

13.4 `bits` — 1 부터 64 비트까지

`type <이름> bits <수> .` 은 그 수만큼의 비트를 가진 정수 타입을 만든다. 폭은 2 의 거듭제곱일 필요가 없다.

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
```

실행 결과

```
$ 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` 는 0 … 1023 이다. 1023 에 1 을 더하면 열 비트에서 넘치므로 멈춘다. 프로토콜의 비트 필드나 압축된 표를 다룰 때 폭이 곧 계약이 된다. 다만 기계의 낱말과 폭이 어긋나면 빠르다는 보장은 없다. 이 언어가 보장하는 것은 열 비트로 셈한 **정확한** 답이다.

13.5 바이트 배치를 못 박는다

구조체의 배치는 보통 처리기가 기계에 맞게 정한다. 선으로 나가는 머리나 장치가 읽는 레지스터처럼 바이트가 **바깥에서도 같은 뜻**이어야 하면 배치를 적는다.

examples/ch13/packed.low

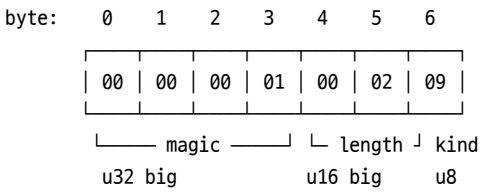
```
module packed .

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
```

실행 결과

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

- `layout packed .` 은 칸 사이에 채움(padding)을 두지 않는다. 칸이 선언 차례로 맞붙는다.
- 칸 뒤의 `big`·`little` 은 바이트 차례를 정한다. 적지 않으면 그 기계의 차례를 따른다.
- `view wire_header b` 는 바이트 슬라이스를 **베끼지 않고** 그 배치의 값으로 읽는다. 길이와 정렬이 맞지 않으면 멈춘다.

`magic 1 · length 2 · kind 9` 를 담은 머리는 바이트로 이렇게 놓인다. `big` 이라 큰 자리가 앞이다.



칸 뒤에는 `rw`·`ro`·`wo` 같은 닿는 법도 적을 수 있다. 장치 레지스터를 비추는 구조체에서 쓰기 전용 칸을 읽으면 번역이 거절된다(30장). 배치를 못 박으면 그만큼 처리기가 고를 여지가 줄어드므로, 필요한 자리에만 박는다.

13.6 배치의 두 방향과 실패하는 뷰

`view` 는 바이트를 그 배치의 값으로 **읽는다**. 반대 방향 — 값을 그 배치의 바이트로 **만드는** 일 — 은 `encode` 가 한다.

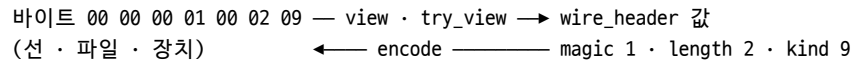
examples/ch13/encoded.low

```
module encoded .
rem run: header_bytes 9

struct wire_header do
  layout packed .
  magic u32 big .
  length u16 big .
  kind u8 .
end

rem `view` 의 반대 방향 --- 값을 그 배치의 바이트로 만든다
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
```

실행 결과

```
$ lowentc --run header_bytes encoded.low 9
header_bytes(9) = 7009
```

`encode wire_header h` 는 칸마다 적힌 바이트 차례(`big`)대로 일곱 바이트를 만든다. 결과 7009 는 길이 7 과 마지막 바이트 `kind` 의 9 를 한데 적은 것이다. 선으로 나갈 머리를 짓는 쪽이 `encode`, 받은 머리를 읽는 쪽이 `view` 다.



`view` 는 길이나 정렬이 맞지 않으면 **멈춘다**. 계약이 깨졌다고 보기 때문이다. 그런데 네트워크에서 온 바이트는 짧은 것이 흔한 일이다. 그런 자리에서는 같은 일을 하되 실패를 값으로 주는 `try_view` 를 쓴다(17장).

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 바깥에서 온 바이트는 짧을 수 있다 --- 멈추는 `view` 대신 `none` 을 주는 `try_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
```

실행 결과

```
$ 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]
```

일곱 바이트면 `kind` 9 를 읽고, 세 바이트면 `none` 을 받아 0 을 돌려준다. 멈추는 `view` 는 이미 확인한 안쪽에서, 값으로 알리는 `try_view` 는 경계에서 쓴다.

13.7 `bitset` — 작은 수의 집합

`bitset <비트 수>` 는 0 부터 그 수 미만까지의 수가 **들어 있는지 없는지**를 담는 집합이다. `bits`(한 정수의 폭)와 이름이 비슷하지만 뜻이 다르다. `bitset` 은 워드의 비트를 다루는 도구가 아니라 집합이고, 워드의 비트는 `bit_and`·`shl` 같은 비트 연산으로 다룬다.

examples/ch13/sets.low

```
module sets .
rem run: overlap 5
```

```
rem run: overlap 3
rem run: flipped

rem 0 … 63 가운데 들어 있는 수의 모음 --- 워드의 비트가 아니라 집합이다
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 집합에서 1 을 지운다 --- 제자리에서 바뀐다
  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 여집합은 폭 안에서만 뜻이 있다 --- 여덟 칸 가운데 하나를 넣었으면 나머지 일곱
fn flipped output u64 .
do
  var a bitset 8 be bitset_new 8 .
  add a 0 .
  return count (complement a) .
end
```

실행 결과

```
$ 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` 가 빈 집합을 만든다. 폭은 번역 시점에 정해진 수다.
- `add a 1 .` 은 넣고 `remove a 1 .` 은 지운다. 둘 다 **제자리에서** 집합을 바꾸는 문장이다. `count a` 는 원소 수다.
- `intersect a b` 는 교집합, `difference a b` 는 `a` 에만 있는 것, `complement a` 는 여집합을 **새 집합으로** 준다.
- `is_subset x y` 는 `x` 가 `y` 에 다 들어 있는지, `is_empty x` 는 비었는지 묻는다.

`overlap 5` 에서 교집합은 {3, 5}, `a` 에만 있는 것은 {1} 이라 211 이 나온다. `complement` 는 폭 안에서만 뜻이 있다. 여덟 칸 집합에 0 하나를 넣었으면 여집합은 나머지 일곱이다. 폭 밖의 수를 넣거나 물으면 멈춘다 — 집합의 범위도 타입의 일부이기 때문이다.

13.8 흩어진 조각을 한 눈으로 — segments

바이트가 한 줄로 이어져 있지 않고 여러 조각에 흩어져 있을 때가 있다. 네트워크에서 받은 조각들이나 링 버퍼의 앞뒤가 그렇다. 그것을 한 자리로 모으려면 베껴야 한다. `segments` 는 베끼지 않고 조각들을 하나의 눈으로 본다.

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 뒤에 놓인 바이트 16 개
  var back mut slice u8 . be some_value mb .
  set (index back 4) 20 .
  set (index back 9) 30 .
  rem 서술자 — (어디부터, 얼마) 짝이 둘: 0 부터 4 개, 8 부터 4 개
  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 복사 없이 두 조각을 한 눈으로 본다
  let ss segments u8 be view_segments back d .
  let p slice u8 be seg ss 1 .
  rem 조각 수 2 에 10 을 곱하고, 둘째 조각의 1 번 바이트(원래 9 번 자리의 30)를 더한다
  return narrow u8 (add (mul (segs ss) 10) (index p 1)) .
end
```

실행 결과

```
$ lowentc --run main pieces.low
main() = 50
```

- `view_segments back d` 는 뒤에 놓인 바이트 `back` 과 서술자 `d` 를 묶어 `segments u8` 타입의 눈을 만든다. 서술자는 (어디부터, 얼마) 짝의 줄이다. 여기서는 0 부터 4 바이트, 8 부터 4 바이트다.
- `segs ss` 는 조각 수 2 를, `seg ss 1` 은 둘째 조각을 평범한 `slice u8` 로 준다. 둘째 조각의 1 번 바이트는 원래 9 번 자리의 30 이다.
- 새 기계 명령은 없다. 묶음 짓기와 칸 읽기와 잘라내기로 낮아지므로 비용이 보인다.

13.9 타입 낱말의 세 갈래

타입 자리에 올 수 있는 낱말은 정본이 닫힌 목록으로 정한다. 그 가운데 일부는 이름만 알고 뜻이 아직 없다.

examples/ch13/notyet.low

```
module notyet .
rem expect: W-NOT-YET
```

```
rem byte 는 이름만 아는 타입 낱말이다 — 뜻이 없어 경고한다. 바이트는 u8 로 적는다
fn first input x byte . output u64 . do
  return 0 .
end

rem 문법 부록 A.9 의 `list` 도 같다 — 목록에는 있었으나 뜻을 정한 조항이 없다
fn count input xs list u64 . output u64 .
do
  return 0 .
end
```

실행 결과

```
$ 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` 는 받아들이되 `W-NOT-YET` 으로 "뜻이 없다" 고 말한다. 조용히 받아 주면 쓰는 사람은 된 줄 안다. 바이트는 `u8` 로 적는다.

갈래	낱말	도구가 하는 일
쓸 수 있다	수 · `bool` · `void` · `slice` · `array` · `segments` · `set` · `stack` · `range` · `vec` · `bitset` · `mask` · `result` · `option` · `ref` · `mut_ref` · `mut` · `owned` · `region` · `cap` · `self`	뜻대로 검사하고 낮춘다
이름만 안다	`byte` · `char` · `str` · `string` · `bytes_view` · `dyn` · `atomic`	`W-NOT-YET` — 뜻이 없다고 말한다
아직 거절	`shared_read` · `lock` · `rwlock`	`E-LOCK-NOTYET` — 흐름끼리 나누는 상태(26장)
한정자	`unsafe_ptr`	타입 앞에 붙는 C 포인터 표시(29장)
정본에 뜻이 없다	`list` · `raw` · `addr` · `rng`	이 판의 도구는 경고 없이 받는다 — 쓰지 않는다

표 13.1 — 타입 낱말 — 쓸 수 있는 것 · 이름만 아는 것 · 거절되는 것

마지막 줄은 이 판의 구멍이다. 네 낱말은 정본의 목록에 있지만 뜻을 정한 조항이 없고, 도구는 아무 말 없이 받아들인다. 개발 저장소에 결함으로 적어 두었다.

13.10 흔한 실수

반례. 넓은 타입의 값을 range 매개변수에 그대로 넘긴다

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` 는 0 … 100 보다 넓다 --- 범위 안이라는 것을 부르는 쪽이 보이지 않았다
  return scale x .
end
```

실행 결과

```
$ 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 은 0 … 100 만 받는다고 서명에 적었다. u64 값은 그보다 넓으므로, 범위 안이라는 것을 부르는 쪽이 보이지 않으면 번역이 E-TYPE-WIDTH 로 거절한다. 거절하는 까닭은 책임의 자리를 정하기 위해서다. 범위를 벗어난 값이 오면 무엇을 할지는 부르는 쪽만 안다. guard 로 범위를 확인해 밖이면 따로 답하고, 안이면 narrow 로 좁혀 넘긴다.

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 범위 밖이면 여기서 따로 답하고, 안이면 좁혀서 넘긴다
  guard le x 100 . else return 255 .
  return scale (narrow u8 x) .
end
```

실행 결과

```
$ lowentc --run call_scale rangewide_fixed.low 40
call_scale(40) = 80
$ lowentc --run call_scale rangewide_fixed.low 400
call_scale(400) = 255
```

반례. 부호 있는 수를 `widen` 으로 부호 없는 타입에 넣는다

examples/ch13/mistake_widensign.low

```
module mistake_widensign .
rem expect: E-WIDEN-SIGN

fn to_count input x i32 . output u64 .
do
  rem ※ 음수는 `u64` 에 자리가 없다 --- 폭이 넓어도 `widen` 은 안 된다
  return widen u64 x .
end
```

실행 결과

```
$ 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
```

`i32` 에서 `u64` 로 가면 폭은 넓어지지만, −1 같은 음수는 `u64` 에 자리가 없다. `widen` 은 **어떤 값도 변하지 않는** 자리에만 쓰므로 `E-WIDEN-SIGN` 이다. 음수가 올 수 없다고 알고 있다면 `cast u64 x` 라고 적어 그 판단을 소스에 남긴다. 음수가 오면 `cast` 가 멈춘다. 음수를 따로 다뤄야 한다면 `guard ge x 0 .` 을 먼저 둔다.

반례. `bool` 을 `cast` 로 수로 바꾼다

examples/ch13/mistake_boolcast.low

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

fn flag input b bool . output u8 .
do
  rem ※ 참이 1 인지는 언어가 아니라 프로그램이 정할 일이다
  return cast u8 b .
end
```

실행 결과

```
$ 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 는 참을 1 로 보지만, 참이 1 인지 0 인지 −1 인지는 프로그램의 약속이지 언어가 정할 일이 아니다. 그래서 `bool` 은 `cast` 의 원천도 대상도 될 수 없고 `E-TYPE-KIND` 로 거절된다. 도구가 알려 주는 대로 `if` 로 "참이면 1, 거짓이면 0" 을 적는다. 그 한 줄이 약속을 소스에 남긴다.

흔한 오해. `type` 으로 이름을 달리하면 단위가 섞이지 않는다

examples/ch13/type_units.low

```
module type_units .
rem run: total 100 100

type meters u64 .
```

```
type feet u64 .

rem 미터와 피트를 더했는데 아무도 막지 않는다 --- `type` 은 다른 이름일 뿐이다
fn total input m meters . input f feet . output meters .
  requires le m 1000000 .
  requires le f 1000000 .
do
  return add m f .
end
```

실행 결과

```
$ lowentc --run total type_units.low 100 100
total(100, 100) = 200
```

`meters` 와 `feet` 는 둘 다 `u64` 의 **다른 이름**일 뿐이라서 더해도 아무도 막지 않는다. 100 미터와 100 피트의 합이 200 미터로 나온다. 섞이면 안 되는 뜻은 `newtype` 으로 가른다. 그러면 `ids.low` 처럼 번역이 `E-TYPE-NOMINAL` 로 막고, 건너는 자리는 `cast` 로 드러난다. `type` 은 긴 이름을 줄이거나 범위를 물려줄 때 쓴다.

흔한 오해. `cast` 로 음수를 부호 없는 타입에 넣으면 C 처럼 큰 수가 된다

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 C 라면 -1 이 4294967295 가 된다 --- Lowent 는 들어가지 않으면 멈춘다
  return cast u32 x .
end
```

실행 결과

```
$ 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)
```

C 에서 `(uint32_t)-1` 은 4294967295 다. 이 감김은 표준이 정한 동작이지만 대개 결함을 숨긴다. Lowent 의 `cast` 는 값을 옮기고, 들어가지 않으면 `E-VM-CAST` 로 멈춘다. 정말로 감기를 원한다면 `narrow_wrap` 처럼 그 뜻을 이름에 지닌 낱말을 쓴다(4장).

13.11 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`type meters u64 .`	같은 타입의 다른 이름	긴 타입을 줄이고 뜻을 남긴다 — 섞임은 막지 않는다

`newtype user_id u64 .`	표현이 같은 새 타입	번호끼리 섞이는 결함을 번역이 막는다 — 실행 비용 없음
`cast user_id n • cast u64 u`	새 타입과 원래 타입 사이를 건넌다	건너는 자리가 코드에서 한곳에 모인다
`input a range 0 100 .`	두 끝을 포함한 범위만 받는다	계약이 서명의 생김새가 된다
`type pct range 0 100 .`	범위를 별칭에 싣는다	여러 op 이 같은 범위를 물려받는다
`cast i32 x`	값이 변할 수 있는 변환 — 들어가지 않으면 멈춤	“여기서 값이 변할 수 있음을 안다” 는 표시
`type ten_bits bits 10 .`	1 … 64 비트 정수	폭이 곧 계약
`layout packed . • magic u32 big .`	채움 없는 배치 · 바이트 차례	바깥과 바이트의 뜻을 맞춘다
`view wire_header b`	바이트를 베끼지 않고 그 배치로 읽는다	길이·정렬이 어긋나면 멈춘다
`try_view wire_header b • encode wire_header h`	실패를 `none` 으로 주는 뷰 · 값을 그 배치의 바이트로	경계에서는 멈추지 않는 쪽
`var a bitset 64 be bitset_new 64 . • add a 1 . • intersect a b`	작은 수의 집합과 그 연산	집합이지 워드의 비트가 아니다
`view_segments back d • segs ss • seg ss i`	흩어진 조각을 베끼지 않고 한 눈으로 · 조각 수 · i 번 조각	모으는 복사를 없앤다
`byte • lock` **같은 타입 낱말**	`W-NOT-YET` · `E-LOCK-NOTYET`	뜻이 없으면 없다고 말한다

표 13.2 — 이름 붙인 타입의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`type` 은 별칭이고 `newtype` 은 표현이 같은 새 타입이며, 둘 사이는 `cast` 로 건넌다. `range lo hi` 는 매개변수의 생김새가 된 계약으로, 부르는 쪽이 지키고 본문은 사실로 쓴다. `cast` 는 값이 변할 수 있는 변환을 적는 낱말이고 들어가지 않으면 멈춘다. `bits N` 은 임의 폭 정수다. `layout packed` 와 바이트 차례 표지, `view` 는 바이트 배치를 바깥과 맞춘다.

제4부 — 계약과 효과

이 부는 Lowent 가 다른 언어와 가장 다른 자리다. 1장에서 세운 목표 — **op 의 머리만 읽고 그 op 이 할 수 있는 일과 못 하는 일을 안다** — 를 실제로 받치는 세 가지를 편다.

계약(op 이 스스로 적는 약속) (14장) — op 이 무엇을 받고 무엇을 돌려주는지 적는다. 누구의 잘못인지를 진단이 가르고, 강제되는 계약은 사실이 되어 실행 중 검사를 지운다.

효과 (15장) — op 이 바깥에 무엇을 하는지 적는다. 효과는 호출을 따라 번지고, 순수함은 최적화의 근거가 된다.

권한 (16장) — 그 일을 누가 허락했는지 적는다. 힘은 전역에 흩어져 있지 않고 인자로 건네받으며, 시작점만 보면 프로그램이 닿을 수 있는 바깥을 안다.

마지막 장(17장)은 셋을 함께 써서, 경계에서는 `result` 로, 안쪽에서는 계약으로 실패의 자리를 정하는 설계를 다룬다.

14 계약 — 적고, 검사받고, 검사를 지운다

이 장의 차례

14.1 계약은 검사되는 약속이다 163
14.2 누구의 잘못인가 163
14.3 계약이 검사를 지운다 165
14.4 모든 원소에 대한 조건 165
14.5 계약에 이름을 준다 166
14.6 지금 판정할 수 있으면 지금 판정한다 166
14.7 계약의 등급 168
14.8 빌드 모드가 남은 검사를 정한다 169
14.9 흔한 실수 170
14.10 이 장의 문법 한눈에 172

먼저 알아야 할 것

2장 첫 프로그램 · 계약은 진입에서 멈춘다
9장 줄 · 길이 조건을 계약으로 적으면 경계 검사가 사라진다
11장 답을 담는 타입 · `errors` 절은 나가는 쪽의 계약이다

돌아보기

9장의 `sum_first` 에서 `requires le n (len a)` . 한 줄이 한 일은 무엇인가? 그리고 색인 자체에 `le` 를 걸면 왜 안 되는가?

> **답.** 검사를 op 진입의 한 번으로 모으고, 반복 안의 `index a i` 경계 검사를 지우게 했다. 색인에 `le` 를 걸면 `i = len a` 가 허락되는데 그것은 한 칸 밖이므로, 색인에는 `lt` 를 쓴다. 이 장은 그 계약이 무엇이고, 누가 지키며, 언제 검사되고 언제 사라지는지를 모두 다룬다.

이 장의 필요성과 맥락

제4부는 이 언어의 이름값을 하는 부분이다. 머리만 읽고 op 이 무엇을 하는지 알게 한다는 목표(1장)는 계약·효과·권한 셋이 받친다. 그 첫째인 계약은 앞의 장들에서 조각으로 여러 번 나왔다 — 진입에서 멈추고, 경계 검사를 지우고, 오류를 약속했다. 이 장은 그 조각들을 하나의 체계로 모은다. 효과(15장)와 권한(16장)이 "무엇을 하는가" 를 다룬다면, 계약은 "무엇을 받고 무엇을 돌려주는가" 를 다룬다.

이 장이 끝나면

`requires·ensures·errors` 가 각각 누구의 책임이고 언제 검사되는지, 깨졌을 때 진단이 어떻게 책임을 가르는지 알게 된다. 계약이 검사를 지우는 원리, 슬라이스의 모든 원소에 대한 조건(`elem_le` 따위), 계약에 이름을 주는 `contract·satisfies` 를 익힌다. 번역할 때 판정되는 위반, 일어날 수 없는 오류의 선언, 계약의 등급(`static·debug·assume`)과 빌드 모드가 남은 검사를 어떻게 다루는지도 보게 된다.

이 장에서 답할 질문

1 `ensures` 는 결국 op 이 자기 자신을 검사하는 것인데, 시험(`test`)과 무엇이 다른가?

14.1 계약은 검사되는 약속이다

계약은 op 이 자기 입력과 출력에 대해 스스로 적는 약속이다. 주석과 달리 검사된다.

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
```

실행 결과

```
$ lowentc --run read_pair pair.low [1,2,3]
read_pair([1,2,3]) = 258
  arg0 (written) = [1,2,3]
```

머리를 소리 내어 읽으면 이렇다. “이 op 은 길이가 2 이상인 바이트 줄을 받아야 하고(requires), 돌려주는 값은 65535 이하임을 약속하며(ensures), 순수하다(fn).” 본문을 열지 않고도 이만큼 안다. ensures 에서 ret 은 돌려주는 값을 가리킨다.

계약은 세 곳에서 쓰인다. 처리기는 계약을 **사실로** 삼아 검사를 지운다. 증명하지 못한 계약은 **실행 중에** 확인되고, 깨지면 멈춘다. 그리고 읽는 사람에게 이 op 을 부르려면 무엇을 지켜야 하는지 말한다.

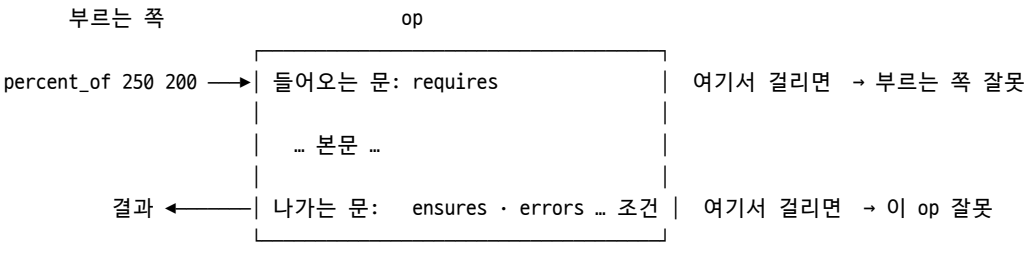
14.2 누구의 잘못인가

계약이 깨지는 자리는 둘이고, 진단이 책임을 가른다.

절	언제 확인되나	깨지면 누구의 잘못인가
requires	들어올 때	**부르는 쪽** — 지켜야 할 조건을 안 지켰다
ensures	나갈 때	**이 op** — 자기 약속을 어겼다
errors … <조건>	나갈 때	**이 op** — 조건이 참인데 그 오류를 내지 않았다

표 14.1 — 계약이 깨졌을 때

계약을 op 의 **드나드는 문 두 개**로 그리면 누구의 잘못인지가 자리로 갈린다.



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 150 을 넘는 값만 자르고, 101 … 150 은 깜빡 잊었다
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
```

실행 결과

```
$ 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` 은 부분이 전체보다 크니 부르는 쪽의 잘못이고, VM 은 “the caller broke the contract” 라고 말한다. `clamp_to_100 120` 은 `n` 이 101 … 150 일 때 자르기를 잊은 op 의 잘못이고, VM 은 “THIS op broke its own promise” 라고 말한다. 프로그램이 멈췄을 때 “내가 잘못 불렀나, 저 op 이 잘못 만들어졌나” 를 진단이 바로 답한다.

> **문. `ensures` 는 결국 op 이 자기 자신을 검사하는 것인데, 시험(`test`)과 무엇이 다른가?**
>
> **답.** 시험은 몇 가지 입력에서 답이 맞는지 본다. `ensures` 는 **모든 호출에서** 약속이 지켜지는지 본다. 시험이 고른 입력 밖에서 약속이 깨져도 `ensures` 는 그 호출에서 멈춘다. 그리고 부르는 쪽은 `ensures` 를 사실로 쓸 수 있다. `clamp_to_100` 을 부른 쪽은 결과가 100 이하임을 믿고 `narrow u8` 의 검사를 지울 수 있다. 개발 저장소의 검증 도구가 기대 출력 없이 계약만으로 op 을 시험할 수 있는 이유이기도 하다(31장).

14.3 계약이 검사를 지운다

처리기는 계약을 사실로 삼아 값이 가질 수 있는 범위를 좁힌다. 그 범위가 안전함을 보이면 넘침·0 나누기·좁히기· 슬라이스 경계 검사를 지운다. 그래서 이 언어에서는 **정직하게 적는 쪽이 빠르다.**

```
fn bare input a u8 . output u8 .
do
  return add a 1 .          rem 넘침 검사가 남는다
end

fn proven input a u8 . output u8 .
  requires le a 200 .
do
  return add a 1 .          rem 검사가 없다 — 201 을 넘을 수 없다
end
```

두 op 은 본문이 같다. 다른 것은 계약 한 줄뿐이고, 그 한 줄이 실행 중 검사 하나를 없앤다. 처리기가 하는 생각은 이렇게 짧다.

```
requires le a 200 .     →  a 는 0 … 200
add a 1                 →  결과는 1 … 201
u8 에 들어가나?          →  201 ≤ 255  ⇒ 넘칠 수 없다  ⇒ 넘침 검사를 지운다
```

검사가 사라진 것이 아니라 진입의 한 번으로 **옮겨진** 것이다. 부르는 쪽이 상수로 부르거나 자기 계약으로 범위를 증명하면 진입 검사마저 사라진다.

중요한 규칙이 하나 있다. **강제되지 않는 계약은 사실로 쓰지 않는다.** 확인 없이 믿고 검사를 지우면, 그것은 빨라진 것이 아니라 틀린 것이다. 이 규칙이 뒤의 등급과 빌드 모드에서 다시 나온다.

14.4 모든 원소에 대한 조건

계약의 조건은 순수한 식이다. 되풀이는 문장이므로 “모든 원소가 9 이하” 를 반복문으로 적을 수 없다. 그래서 그 말을 하는 낱말이 따로 있다.

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
```

실행 결과

```
$ 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` 는 모든 원소가 9 이하라는 뜻이다. `elem_lt`·`elem_gt`·`elem_ge` 도 있다. 이 조건은 진입에서 한 번 확인되고, 본문의 산술은 원소가 9 이하라는 사실을 쓴다.

14.5 계약에 이름을 준다

여러 op 이 같은 조건을 요구하면 그 조건에 이름을 준다.

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
```

실행 결과

```
$ lowentc --run half named.low 10
half(10) = 5
$ lowentc --run third named.low 10
third(10) = 3
```

`contract positive do … end` 가 이름 붙은 계약이고, op 은 머리 맨 앞의 `satisfies positive .` 로 그것을 갖춘다. 같은 조건을 여러 곳에 손으로 되풀이하면 하나를 고치고 다른 하나를 잊는다. 이름을 주면 고칠 자리가 하나가 된다. `satisfies` 가 머리의 **맨 앞**에 오는 것은 이 op 이 무엇인지를 먼저 말하기 때문이다(3장).

14.6 지금 판정할 수 있으면 지금 판정한다

부르는 자리가 상대의 `requires` 를 어기고 양쪽이 모두 상수이면, 실행을 기다리지 않고 번역에서 거절된다.

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
```

실행 결과

```
$ 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)
```

진단이 적은 사연대로, 이것은 한때 초록으로 번역되어 실행 중에야 멈췄다. 맞지 않는 비트 예산을 지닌 프로그램이 실행할 수 있는 모습으로 배포되었다. 답이 지금 나는 계약은 지금 말한다.

반대쪽 실수도 거절된다. `requires` 가 이미 배제한 조건에 오류를 선언하면, 그 오류는 결코 나지 않는다.

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
```

실행 결과

```
$ 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)
```

`requires ne b 0 .` 가 있으니 `errors by_zero eq b 0 .` 은 영영 참이 될 수 없다. 이 선언이 남아 있으면 부르는 쪽은 `by_zero` 를 다루는 코드를 쓰고, 그 코드는 영영 돌지 않는다. 돌지 않는 코드는 시험되지 않고, 시험되지 않는 코드는 언젠가 틀린다. 진단은 둘 중 하나를 지우라고 한다 — 부르는 쪽에 책임을 지우든(`requires`), op 이 스스로 처리하든(`errors`).

> **흔한 오해. 계약은 많이 적을수록 안전하다**
>
> 적은 계약이 서로 겹치거나 일어날 수 없는 경우를 약속하면 틀린 그림이 된다. 계약은 **사실**이어야 한다. `requires` 와 `errors` 가 같은 조건을 두고 다투는 것은 누가 책임지는지를 흐리는 일이고, 그래서 거절된다. 좋은 계약은 짧고, 책임이 한 곳에 있다.

14.7 계약의 등급

계약 절 하나에 등급을 붙여 그 조건을 언제 무엇으로 다룰지 정할 수 있다. `requires <등급> <조건> .` 꼴이다.

등급	언제 보나	뜻
(없음)	할 수 있을 때	증명할 수 있으면 증명하고, 못 하면 빌드 모드대로 남긴다
`static`	번역할 때	처리기가 증명해야 한다. 못 하면 번역이 실패한다
`debug`	돌 때	빌드 모드가 검사를 두는 동안만 본다
`assume`	보지 않는다	적어 두기만 한다. 사실로 쓰지 않는다

표 14.2 — 계약의 등급

`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
```

실행 결과

```
$ 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)
```

같은 조건을 두 가지로 적었다. `bump_checked` 는 진입에서 250 을 거절하고, 그 뒤 `add a 1` 의 넘침 검사를 지운다. `bump_assumed` 는 진입에서 검사하지 않는다. 대신 조건을 사실로 쓰지도 않으므로 넘침 검사가 **남는다**. 그래서 255 를 주면 계약이 아니라 넘침(`E-VM-OVERFLOW`)으로 멈춘다.

`assume` 은 아직 처리기가 증명할 수 없지만 사람은 아는 것을 적어 두는 자리다. 적어 두면 읽는 사람이 알고, 나중에 처리기가 자라면 `static` 으로 올릴 수 있다. 지켜지지 않는 조건을 사실로 삼으면, 조건이 거짓일 때 지우지 말았어야 할 검사를 지운 채 옳다고 믿게 된다. 그래서 `assume` 은 사실이 아니다.

14.8 빌드 모드가 남은 검사를 정한다

증명된 계약은 어느 모드에서도 검사가 남지 않는다. 빌드 모드가 정하는 것은 **증명하지 못한** 계약의 처분이다. 모드는 소스에 `build <모드> .` 로 적는다.

모드	남은 계약 검사
`debug`	두고, 깨지면 무엇이 깨졌는지 말하며 멈춘다
`test`	`debug` 처럼 두고, 시험도 함께 짓고 돌린다
`release_safe`	두되 메시지 없이 멈춘다
`release_fast`	**없앤다**. 계약이 깨진 채로 진행할 수 있다

표 14.3 — 빌드 모드 넷

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
```

실행 결과

```
$ lowentc --run bump fast.low 10
bump(10) = 11
$ lowentc --run bump fast.low 250
bump(250) = 251
```

`build release_fast .` 아래에서 `bump 250` 은 멈추지 않고 251 을 낸다. 같은 소스가 `debug` 에서는 멈춘다. `release_fast` 는 계약이 참임을 **사람이 약속하는** 신뢰 경계를 하나 더 여는 것이다. 이 언어가 미정의 동작을 없앴다고 말하는 범위가 "안전한 부분집합" 으로 한정되는 이유 가운데 하나다. 속도를 위해 검사를 없애는 것은 고를 수 있는 일이지만, 고른다는 사실이 소스에 남아야 한다. 그래서 모드는 명령 줄 깃발이 아니라 `build` 문장이다.

> **실제 사례. 처리기가 만들 수 없는 진입 검사**
>
> 계약의 모든 식을 진입 검사로 만들 수 있는 것은 아니다. 이 판의 처리기는 `requires` 에서 `이름 비교 상수`, `len`, `elem_*`, 칸 경로 같은 꼴을 진입 검사로 만들고, 만들 수 없는 식을 만나면 `W-CONTRACT-IGNORED` 로 알

린다. 이 책을 쓰며 확인한 바로는 ensures 에 식이 든 조건(ensures le (mul ret 2) n .)은 경고 없이 검사되지 않았다. 계약이 강제되지 않으면 사실로도 쓰이지 않으므로 틀린 최적화는 생기지 않지만, 약속이 지켜지는지 확인되지 않는다는 사실은 알려야 한다. 이런 자리는 처리기가 알려야 하는 곳이므로, 이 판의 결함이다.

14.9 흔한 실수

반례. ensures 에서 돌려주는 값을 result 라고 부른다

examples/ch14/mistake_result.low

```
module mistake_result .
rem expect: E-ENS-UNDEF

fn clamp input a u8 . output u8 .
  rem ✘ 돌려주는 값의 이름은 `result` 가 아니라 `ret` 이다
  ensures le result 100 .
do
  guard le a 100 . else return 100 .
  return a .
end
```

실행 결과

```
$ 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)
```

돌려주는 값의 이름은 ret 이다. result 는 타입 이름(result u8 e)에 이미 쓰이므로 값 이름으로 겹쳐 쓰지 않는다. 없는 이름을 가리키는 ensures 는 아무것도 검사하지 않으면서 처리기가 결과 범위를 끌어내는 데 쓰일 수 있으므로, 경고가 아니라 오류 (E-ENS-UNDEF)다.

반례. 색인의 전제를 le 로 적는다

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` 는 `i = len xs` 를 허락한다 --- 그 칸은 한 칸 밖이다
  requires le i (len xs) .
do
  return index xs i .
end
```

실행 결과

```
$ 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)
```

길이가 3 인 줄의 칸 번호는 0, 1, 2 다. `requires le i (len xs)` 는 `i = 3` 도 허락하므로 계약은 통과하고, 그다음 `index` 가 한 칸 밖을 읽으려다 `E-VM-BOUNDS` 로 멈춘다. 계약이 틀리면 멈추는 자리가 계약에서 본문으로 옮겨 가고, 진단이 "부르는 쪽의 잘못" 대신 "경계 밖" 을 말한다. 색인에는 `lt` 를 쓴다. 개수(`n` 개를 읽는다)에는 `le` 가 맞다.

반례. 함께 만족할 수 없는 전제를 적는다

examples/ch14/mistake_contradict.low

```
module mistake_contradict .
rem expect: E-CONTRACT-UNSAT

rem ✘ 두 조건을 함께 만족하는 값이 없다 --- 어떻게 불러도 멈춘다
fn pick input a u8 . output u8 .
  requires le a 100 .
  requires ge a 200 .
do
  return a .
end
```

실행 결과

```
$ 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
```

100 이하이면서 200 이상인 `u8` 은 없다. 그러니 이 op 은 어떻게 불러도 진입에서 멈춘다. 대개 한 줄의 `le·ge` 를 거꾸로 적었거나, 고치다가 옛 줄을 지우지 않은 것이다. 일어날 수 없는 오류 선언(`E-CONTRACT-DEAD`)과 같은 자리이므로 `E-CONTRACT-UNSAT` 으로 거절한다 — 어떤 인자도 만족하지 못하는 계약은 강한 계약이 아니라 죽은 op 이다.

흔한 오해. `requires` 는 사용자 입력을 검증하는 도구다

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 계약: 부르는 쪽이 이미 확인한 값만 받는다 --- 어기면 프로그램이 멈춘다
fn age_strict input a u8 . output u8 .
```

```
  requires le a 150 .
do
  return a .
end

rem 바깥에서 온 값은 틀릴 수 있다 --- 틀림을 값으로 돌려준다
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
```

실행 결과

```
$ 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
```

계약이 깨지면 **프로그램이 멈춘다**. 계약은 "부르는 쪽이 이미 확인했다" 는 약속이라서, 깨졌다는 것은 코드에 결함이 있다는 뜻이다. 사용자가 200 살이라고 입력한 것은 결함이 아니라 흔히 일어나는 일이다. 바깥에서 들어온 값은 `age_checked` 처럼 확인해서 틀림을 `result` 로 돌려주고, 확인이 끝난 값만 `age_strict` 같은 계약 있는 op 에 넘긴다. 확인한 뒤에 부르므로 `check_age` 안의 계약 검사는 처리기가 지운다.

14.10 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`requires le a 200 .`	들어올 때의 조건 — 부르는 쪽의 책임	검사를 진입의 한 번으로 모으고 본문의 검사를 지운다
`ensures le ret 100 .`	나갈 때의 약속 — 이 op 의 책임	`ret` 은 돌려주는 값 — 부르는 쪽이 사실로 쓴다
`errors too_big gt a 200 .`	이 조건이면 이 오류를 낸다는 약속	나가는 쪽의 계약 — 오류도 약속한다
`requires elem_le ds 9 .`	모든 원소에 대한 조건	계약은 식이라 반복문을 쓸 수 없다

`contract positive do … end`	계약에 이름을 준다	같은 조건을 고칠 자리가 하나가 된다
`fn half satisfies positive . …`	이름 붙은 계약을 갖춘다(머리 맨 앞)	op 이 무엇인지 먼저 말한다
`requires static … • debug • assume`	계약의 등급	언제 무엇으로 볼지 절마다 정한다 — `assume` 은 사실이 아니다
`build release_fast .`	증명하지 못한 계약 검사를 없앤다	속도를 고른 사실이 소스에 남는다

표 14.4 — 계약의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`requires` 는 부르는 쪽, `ensures` 와 `errors` 는 op 의 책임이고, 진단이 누구의 잘못인지 말한다. 강제되는 계약은 사실이 되어 넘침·경계·0 나누기 검사를 지운다. `elem_*` 는 모든 원소에 대한 조건이고, `contract` 와 `satisfies` 는 계약에 이름을 준다. 상수끼리의 위반은 번역에서, 일어날 수 없는 오류 선언도 번역에서 거절된다. `assume` 은 사실이 아니며, 빌드 모드는 증명하지 못한 계약의 처분을 정하고 `release_fast` 는 그 검사를 없앤다.

15 효과 — op 이 세상에 남기는 자국

이 장의 차례

15.1 효과는 닫힌 낱말이다 174
15.2 효과는 부르는 쪽으로 번진다 175
15.3 적어 놓고 안 하면 알린다 177
15.4 효과 줄은 집합이다 177
15.5 via — 타입 인자가 효과를 정한다 178
15.6 순수함이 허락하는 것 179
15.7 흔한 실수 179
15.8 이 장의 문법 한눈에 183

먼저 알아야 할 것

5장 op · `fn` 은 순수하고 `proc` 의 `effects` 는 좁히는 절이다
14장 계약 · 강제되지 않는 약속은 사실로 쓰지 않는다

돌아보기

5장에서 순수한 `helper` 가 스스로 출력하지 않고 `say` 를 부르기만 했는데도 거절되었다. 왜인가?

> **답.** 효과는 부르는 쪽으로 번지기 때문이다. `say` 가 `io` 를 내므로 그것을 부르는 `helper` 도 `io` 를 내는 것이 되고, 순수한 `fn` 은 효과를 낼 수 없다. 이 장은 그 효과가 무엇으로 이루어지고, 선언과 실제가 어느 쪽으로 어긋날 때 무엇이 일어나는지를 다룬다.

이 장의 필요성과 맥락

계약(14장)이 값의 약속이라면 효과는 **행동**의 약속이다. 파일을 읽는가, 메모리를 받는가, 멈출 수 있는가, 다른 흐름을 기다리는가. 이 물음의 답이 머리에 없으면 호출자는 본문을 끝까지 따라가야 한다. Lowent 는 효과를 닫힌 낱말의 집합으로 적게 하고 컴파일러가 양쪽으로 검사한다. 권한(16장)은 그 효과를 **누가 허락했는지**를 적는 짝이므로, 효과를 먼저 세운다.

이 장이 끝나면

효과가 닫힌 원자의 집합이라는 것과 원자마다 무엇을 뜻하는지 알게 된다. 효과가 호출을 따라 번지고, 적은 것보다 많이 하면 거절되며 적어 놓고 안 하면 경고된다는 것을 익힌다. 효과 줄이 목록이 아니라 집합이라 생기는 규칙, `concurrent` 가 `wait` 을 딸고 오는 닫힘, 타입 인자가 효과를 정하게 하는 `via` 를 보게 된다. 마지막으로 순수함이 컴파일러에게 무엇을 허락하는지 이해하게 된다.

이 장에서 답할 질문

1 로그 한 줄을 남기려고 순수한 op 전체를 `proc` 으로 바꿔야 하는가?

15.1 효과는 닫힌 낱말이다

효과는 op 이 바깥에 주는 영향이다. 저자가 새 효과를 만들 수는 없다. 목록은 언어가 정해 두었다.

원자	뜻
none	아무 효과도 없다. 바닥이다
io	바깥과 자료를 주고받는다(파일·연결·표준 입출력)
alloc	**고정 창**에서 메모리를 얻거나 돌려준다
heap	**자라는 뿌리**에서 메모리를 얻는다. 운영체제가 있는 기계에만 있다
state	모듈이나 액터가 지닌 상태, 호출자의 저장소를 고친다
panic	프로그램을 멈출 수 있다
atomic	나눌 수 없는 읽기·쓰기를 한다
concurrent	완결이 다른 실행 흐름의 진행에 달려 있다
wait	기다린다. 누구의 도움 없이도 언젠가 깨어난다
lock	자물쇠를 잡는다
device	장치를 직접 건드린다
unsafe	언어가 검사할 수 없는 일을 한다
page_fault · blocking · cancel · detach	흐름이 멈추거나 끊기는 일들 — 아직 이를 내는 원시 연산이 없다

표 15.1 — 효과 원자

`io` `alloc` `heap` `state` `panic` `atomic` `concurrent` `wait` `unsafe` 아홉은 그것을 실제로 내는 원시 연산이 있고 컴파일러가 강제한다 — 몸이 그 일을 하면 적어야 하고, 적어 놓고 안 하면 알린다. 나머지 여섯(`lock` `device` `page_fault` `blocking` `cancel` `detach`)은 그 일을 하는 원시 연산이 아직 없다. 그래서 선언으로 받아 부르는 쪽으로 **번지게**는 하지만, 몸이 정말 그 일을 하는지는 묻지 못한다. 목록에 없는 낱말은 거절된다.

examples/ch15/undef.low

```
module undef .
rem expect: E-EFFECT-UNDEF

proc checked input x u64 . output u64 . effects crash .
do
  return x .
end
```

실행 결과

```
$ 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
```

오타를 조용히 받아 주면 그 op 은 효과가 없는 것으로 읽힌다. 진단의 말대로 “a typo here silently declares the op PURE” 가 되는 자리를 막는다.

15.2 효과는 부르는 쪽으로 번진다

op 을 부르면 그 op 의 효과가 부르는 쪽 효과에 더해진다. 적은 것보다 많은 효과를 내면 거절된다.

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
```

실행 결과

```
$ 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` 는 `effects panic .` 만 적었는데 `log_line` 을 부르므로 `io` 도 낸다. 진단은 "`effects` 에 더하거나, 그것이 필요한 것을 부르지 말라" 고 한다. 반대로 효과가 번지지 않는 층도 분명하다.

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
```

실행 결과

```
$ lowentc --run main layered.low
2
main() = 0
```

`celsius_to_f` 는 순수한 `fn` 이라 어디서든 부를 수 있다. `print_digit` 은 `io` 를 내므로 `io` 를 선언한 `main` 에서만 부를 수 있다. 계산하는 층과 바깥과 닿는 층이 머리에서 갈린다. 좋은 설계는 순수한 층을 넓게, 효과 있는 층을 얇게 둔다.

> **문. 로그 한 줄을 남기려고 순수한 op 전체를 `proc` 으로 바꿔야 하는가?**
>
> **답.** 그렇다. 로그를 남기는 op 은 바깥에 흔적을 남기므로 순수하지 않고, 그 사실이 머리에 보여야 한다. 대신 흔한 해법이 있다. 계산은 순수하게 두고 **결과를 돌려주며**, 로그는 그 결과를 받은 바깥 층에서 남긴다. 효과를 바깥으로 밀어내는 이 모양은 시험하기도 쉽다 — 순수한 층은 권한 없이 `--run` 과 `test` 로 바로 확인된다.

15.3 적어 놓고 안 하면 알린다

선언만 하고 내지 않는 효과도 문제다. 선언한 효과는 부르는 쪽이 감당해야 하는 비용이기 때문이다.

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
```

실행 결과

```
$ 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` 은 멈출 수 있다고 적었지만 멈추지 않는다. 그러면 이 op 을 부르는 모든 쪽이 `panic` 을 적어야 하고, 순수한 `fn` 은 부를 수 없게 된다. 하는 일을 부풀려 말한 것이다. 진단은 지우라고 하고, 다음 판에서 실제로 그 효과를 낼 예정이라면 그렇다고 적으라고 한다.

> **흔한 오해. 효과를 넉넉하게 적어 두면 나중에 고칠 일이 줄어든다**
>
> 넉넉한 효과는 부르는 쪽 전체로 번진다. `panic` 하나를 미리 적으면 그 op 을 부르는 모든 op 이 `panic` 을 적어야 하고, 순수해야 할 계산이 순수함을 잃는다. 순수함을 잃은 op 에는 차례 바꾸기·기억해 두기 같은 최적화도 허락되지 않는다. 효과는 실제로 하는 만큼만 적는다.

15.4 효과 줄은 집합이다

효과 줄은 목록이 아니라 **집합**이다. 같은 낱말을 두 번 적으면 거절된다.

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
```

실행 결과

```
$ 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)
```

두 번째 `panic` 은 첫 번째가 하지 않은 말을 하나도 하지 않는다. `none` 은 다른 원자와 함께 올 수 없다.

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
```

실행 결과

```
$ 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)
```

효과가 있으면서 없을 수는 없다. 효과를 적든지, 없다고 적든지 하나다. 그리고 `fn` 에는 `effects none` 도 적지 않는다(5장).

어떤 효과는 다른 효과를 딸고 온다. 지금 규칙은 하나다 — `concurrent` 를 적으면 `wait` 을 함께 적은 것이 된다. 다른 흐름의 진행을 기다려야 한다면 중단될 수도 있기 때문이다. 그래서 두 집합을 견줄 때는 이 닫힘을 적용한 뒤에 견준다. `concurrent` 를 적으면 `wait` 을 따로 적지 않아도 되지만, `wait` 만 적고 `concurrent` 한 일을 하면 거절된다.

15.5 via — 타입 인자가 효과를 정한다

제네릭 컨테이너는 곤란한 처지에 놓인다. 어떤 할당기를 받느냐에 따라 효과가 달라지기 때문이다. 빌린 바이트 위에서 깎는 할당기로 열면 `state` 뿐이고, 자라는 힘을 딛는 할당기로 열면 `heap` 이 선다. 모든 할당기가 낼 수 있는 가장 큰 효과를 늘 적으면 거짓이 되고, 가장 작은 것만 적으면 **숨은 할당**이 된다.

```
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 .
```

표준 라이브러리 `vecgen` 의 `append` 머리다. `effects state via a .` 는 "타입 `a` 의 op 들이 적은 할당 계열 효과(`alloc·heap·atomic`)도 이 op 의 선언이다" 라는 뜻이다. 단형화된 인스턴스마다 그 타입이 효과를 정한다. 제네릭 컨테이너는 34장에서 다시 만난다.

15.6 순수함이 허락하는 것

`fn`, 곧 효과가 `none` 인 op 은 세 가지를 만족한다.

- 같은 입력에 언제나 같은 결과를 낸다.
- 부르는 차례를 바꾸어도 프로그램의 결과가 달라지지 않는다.
- 결과를 쓰지 않는다면 부르지 않아도 된다.

이 세 성질이 컴파일러에게 최적화를 허락한다. 같은 인자로 두 번 부른 순수 호출은 한 번으로 줄일 수 있고 (공통 부분식 제거), 결과를 기억해 둘 수 있으며(memoise), 쓰이지 않는 호출은 지울 수 있고, 서로 기대지 않는 호출은 차례를 바꾸거나 나누어 돌릴 수 있다. 이 최적화들이 프로그램의 뜻을 바꾸지 않는다는 것은 Coq 로 증명되어 있다(44장). 다만 호출 경계와 멈추는 시점은 그 증명 밖이다.

> **실제 사례. 효과 선언이 증명의 전제가 되는 자리**
>
> 효과 체계의 건전성 정리는 "선언한 효과가 실제로 일어나는 효과를 덮는다" 는 것이다. 이 정리가 있어야 효과에 기댄 최적화의 정리가 선다. 그래서 컴파일러는 선언보다 **많이** 하는 코드를 거절하고(정리의 전제), 선언보다 **적게** 하는 코드는 경고한다(정리에는 해가 없지만 부르는 쪽에 거짓 비용을 떠넘긴다). 두 방향의 무게가 다른 이유가 여기에 있다.

15.7 흔한 실수

> **반례. 순수한 `fn` 에서 `panic` 을 쓴다**

examples/ch15/mistake_fnpanic.low

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

rem ✘ 순수한 `fn` 이 `panic` 을 쓴다 --- 멈추게 하는 것도 효과다
fn nonzero input x u64 . output u64 .
do
  if eq x 0 . do panic "zero" . end
  return x .
end
```

실행 결과

```
$ 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
```

프로그램을 멈추게 하는 것도 바깥에 남는 자국이다. 부르는 쪽이 이 op 을 기억해 두거나 차례를 바꾸면 멈춤이 일어나는 때가 달라지므로, `fn` 은 `panic` 을 쓸 수 없고 `E-EFFECT-CALC` 로 거절된다. `fn` 에 `effects`

panic . 을 붙이는 것도 같은 코드로 거절된다 — 순수한가 아닌가는 fn·proc 이라는 낱말이 이미 말하기 때문이다. 고치는 길은 둘이다. 멈춤을 효과로 적은 proc 으로 바꾸거나, 그 조건을 부르는 쪽의 책임인 계약으로 옮긴다.

examples/ch15/fnpanic_fixed.low

```
module fnpanic_fixed .
rem run: nonzero_checked 5
rem trap: nonzero_strict 0

rem 고친 길 1 --- 멈춤을 효과로 적은 `proc`
proc nonzero_checked input x u64 . output u64 . effects panic .
do
  if eq x 0 . do panic "zero" . end
  return x .
end

rem 고친 길 2 --- 조건을 계약으로 옮기면 `fn` 으로 남는다
fn nonzero_strict input x u64 . output u64 .
  requires ne x 0 .
do
  return x .
end
```

실행 결과

```
$ 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
```

두 번째 길이 대개 낫다. nonzero_strict 는 순수함을 지키고, 0 을 넘긴 잘못이 부르는 쪽에 있다고 진단이 말한다.

반례. fn 이 mut 로 받은 줄에 쓴다

examples/ch15/mistake_fnmutwrite.low

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

rem ✘ `fn` 이 `mut` 로 받은 호출자의 줄에 쓴다
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
```

실행 결과

```
$ 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)
```

`clear` 가 쓴 0 은 호출자의 줄에 남는다. 호출자가 보기에 이것은 바깥이 바뀐 것이고, 그런 op 은 기억해 두거나 지우거나 차례를 바꿀 수 없다. 그래서 `E-EFFECT-PURITY` 다. 머리를 `proc clear … effects state .` 로 바꾼다. 진단이 덧붙인 말대로 op 안의 지역을 고치는 것은 순수함을 깨지 않는다(아래 오개념).

반례. 효과 사이에 쉼표를 넣는다

examples/ch15/mistake_effcomma.low

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

rem ✘ 효과 사이에 쉼표를 넣었다 --- 낱말을 띄어 적기만 한다
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
```

실행 결과

```
$ 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.
```

목록을 쉼표로 가르는 습관이 여기서는 `E-VOCAB-REMOVED` 가 된다. 쉼표는 한때 문법에 있었으나 한 줄을 이어 적는 데에만 쓰였고, 지금은 줄바꿈이 형식을 닫지 않으므로 필요 없어 빠졌다. `effects io panic .` 처럼 띄어 적는다. 형식은 마침표에서 끝난다.

흔한 오해. `var` 와 `set` 을 쓰면 순수하지 않다

examples/ch15/pure_local.low

```
module pure_local .
rem run: sum_to 10

rem `var` 와 `set` 을 써도 순수하다 --- 바뀌는 것이 이 op 안에서만 보이기 때문이다
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
```

실행 결과

```
$ lowentc --run sum_to pure_local.low 10
sum_to(10) = 55
```

`sum_to` 는 지역 변수 둘을 계속 고치지만 `fn` 이다. 바뀌는 저장소가 이 op 안에만 있고 호출자는 결과 55 만 본다. 같은 입력에 언제나 같은 답을 내고, 부르지 않아도 바깥에 남는 것이 없다. 순수함은 "안에서 아무것도 바꾸지 않는다" 가 아니라 "바깥에서 보이는 자국이 없다" 이다.

흔한 오해. `fn` 은 절대 멈추지 않는다

examples/ch15/fn_can_stop.low

```
module fn_can_stop .
rem run: first [7,8]
rem trap: first []

rem 순수한 `fn` 이지만 빈 줄을 받으면 멈춘다 --- 경계 검사의 멈춤은 op 이 낸 효과가 아니
다
fn first input xs slice u8 . output u8 .
do
  return index xs 0 .
end
```

실행 결과

```
$ 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)
```

`fn` 이 쓸 수 없는 것은 `panic` **효과**다. 빈 줄의 첫 칸을 읽는 것처럼 계약이나 경계가 깨져 처리기가 멈추는 것은 op 이 한 일이 아니라 잘못 불린 결과로 본다(정본 6.5.9). 그래서 순수한 `first` 도 `[]` 를 받으면 멈춘다. 멈추지 않게 하려면 `requires ge (len xs) 1 .` 로 책임을 부르는 쪽에 드러내거나, `option` 을 돌려준다.

반례. 드문 효과를 적은 op 을 순수한 `fn` 에서 부른다

examples/ch15/mistake_blocking.low

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

rem blocking — 실행 흐름을 붙잡아 둘 수 있다고 머리에 적는다
proc wait_for_device output u64 . effects blocking .
do
  return 0 .
```

```
end

rem ✘ 순수한 fn 이 흐름을 붙잡을 수 있는 op 을 부른다 — 이 판에서는 통과해 버린다
fn ready output u64 . do
  return wait_for_device .
end
```

실행 결과

```
$ 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` 은 "흐름을 붙잡아 둘 수 있다" 이므로, 그런 op 을 부르는 `fn` 은 순수하지 않다. 그래서 `E-EFFECT-CALC` 로 거절된다. 2026-09-16 까지는 `lock·device·page_fault·blocking·cancel·detach` 여섯이 부르는 쪽으로 **번지지 않아** 통과했다 — 특히 장치를 직접 건드리는 `device` 가 순수 함수 뒤에 숨었다. 같은 모양에 `wait` 를 적으면 예전에도 거절됐다.

examples/ch15/blocking_wait.low

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

rem wait 는 번진다 — 같은 모양이 E-EFFECT-CALC 로 거절된다
proc wait_for_device output u64 . effects wait .
do
  return 0 .
end

fn ready output u64 . do
  return wait_for_device .
end
```

실행 결과

```
$ 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
```

이 여섯은 아직 그 일을 하는 원시 연산이 없어서 "선언했는데 안 한다"(`W-EFFECT-OVER`)로는 묻지 않는다 — 물을 몸이 없기 때문이다. 전파만 한다.

15.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`fn …`	효과 없음 — `effects` 절을 적지 않는다	순수함이 낱말 하나로 보인다
`proc … effects io panic .`	이 op 이 낼 수 있는 효과의 집합	머리만 읽고 무엇을 하는지 안다

`proc …` **(절 없음)**	좁히지 않았다 — 무엇이든 낼 수 있다	좁히는 것은 저자의 선택
효과 원자 `io`·`alloc`·`heap`·`state`·`panic`·…	언어가 정한 닫힌 목록	오타가 조용히 “순수” 가 되지 않게
`effects panic panic .` · `none panic`	거절(`E-EFFECT-DUP` · `E-EFFECT-NONE-MIX`)	집합이라 겹침과 모순이 없다
적은 것보다 많이 함 · 적고 안 함	오류(`E-EFFECT`) · 경고(`W-EFFECT-OVER`)	증명의 전제 · 부르는 쪽의 거짓 비용
`concurrent`	`wait` 을 딸고 온다	다른 흐름을 기다리면 멈출 수 있다
`effects state via a .`	타입 인자 `a` 의 할당 계열 효과를 물려받는다	할당기마다 효과가 정확해진다

표 15.2 — 효과의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

효과는 닫힌 원자의 집합이고, 목록 밖의 낱말은 거절된다. 효과는 호출을 따라 번지며, 적은 것보다 많이 하면 거절되고 적어 놓고 안 하면 경고된다. 효과 줄은 집합이라 중복과 `none` 섞기가 거절되고, `concurrent` 는 `wait` 을 딸고 온다. `via` 는 타입 인자가 할당 계열 효과를 정하게 한다. 순수함은 차례 바꾸기·기억하기·지우기 같은 최적화를 허락한다.

16 권한 — 건네받는 힘

이 장의 차례

16.1 권한의 종류 185
16.2 권한은 사슬을 타고 내려간다 186
16.3 효과와 권한은 짝이다 187
16.4 있음이 아니라 종류다 189
16.5 시작점이 받는 것 190
16.6 흔한 실수 191
16.7 이 장의 문법 한눈에 194

먼저 알아야 할 것

2장 첫 프로그램 · 출력하려면 `input out cap io .` 를 받는다
15장 효과 · 효과는 닫힌 원자의 집합이고 호출을 따라 번진다

돌아보기

15장에서 효과 줄은 op 이 **무엇을 하는지**를 적는다고 했다. 그렇다면 `effects io` 만 보고 알 수 없는 것은 무엇인가?

> **답. 무엇과** 입출력을 하는지다. 표준출력에 쓰는지, 파일을 여는지, 연결을 여는지는 `io` 한 낱말로 갈리지 않는다. 이 장의 권한이 그것을 좁혀 준다 — `input fs cap file_system .` 을 받은 op 은 파일은 열 수 있어도 연결은 열 수 없다.

이 장의 필요성과 맥락

대부분의 언어에서 어떤 함수든 표준출력에 쓰고, 파일을 열고, 시계를 읽을 수 있다. 그 힘이 전역에 흩어져 있어서 "이 라이브러리가 네트워크에 닿는가" 를 알려면 소스 전체를 읽어야 한다. Lowent 에는 그런 주변 권한(ambient authority)이 없다. 힘은 **값처럼 인자로 건네받는다.** 효과(15장)와 권한은 같은 일을 두 쪽에서 적어 서로를 검사하는 짝이고, 이 장으로 제4부의 세 기둥(계약·효과·권한)이 선다.

이 장이 끝나면

권한의 종류와 각 종류가 여는 일을 알게 된다. 권한이 호출 사슬을 따라 이름으로 넘어가고, 시작점만 보고도 프로그램이 닿을 수 있는 바깥을 안다는 것을 익힌다. 효과와 권한의 짝표, 권한 없이 효과를 선언하거나 다른 종류를 건네거나 권한을 적지 않고 쓸 때의 진단을 보게 된다. 시작점이 받을 수 있는 권한의 목록과, 권한이 실행 중 비용이 없다는 것도 이해하게 된다.

이 장에서 답할 질문

1 권한을 인자로 넘기면 실행할 때 인자가 하나씩 늘어 느려지지 않는가?

16.1 권한의 종류

권한은 `input <이름> cap <종류> .` 로 받는다. 종류가 그 권한으로 할 수 있는 일을 정한다.

권한	여는 일
cap io	표준 입출력
cap file_system	파일과 디렉터리
cap net	연결을 만들고 주고받는다
cap tty	터미널의 크기와 모드
cap clock	시각을 묻는다
cap random	운영체제의 엔트로피
cap args · cap env	프로그램 인자 · 환경 변수
cap allocator	고정 창에서 메모리를 얻는다(자라지 않는다)
cap heap	자라는 뿌리에서 메모리를 얻는다(운영체제가 있는 기계만)
cap atomic	원자적 연산
cap mmio	장치 레지스터
cap c	C 함수를 부른다
cap machine	기계 명령을 직접 낸다

표 16.1 — 처리기가 뜻을 정해 둔 권한의 종류

종류를 나누는 까닭은 **최소 권한**이다. 권한이 하나뿐이면 파일을 읽으려고 받은 자격이 연결도 열 수 있게 되고, 그러면 op 의 머리가 "이 op 이 무엇을 할 수 있는가" 를 더는 말해 주지 못한다.

16.2 권한은 사슬을 타고 내려간다

권한을 다시 넘길 때 특별한 표기는 없다. 받은 이름을 다른 인자처럼 적는다.

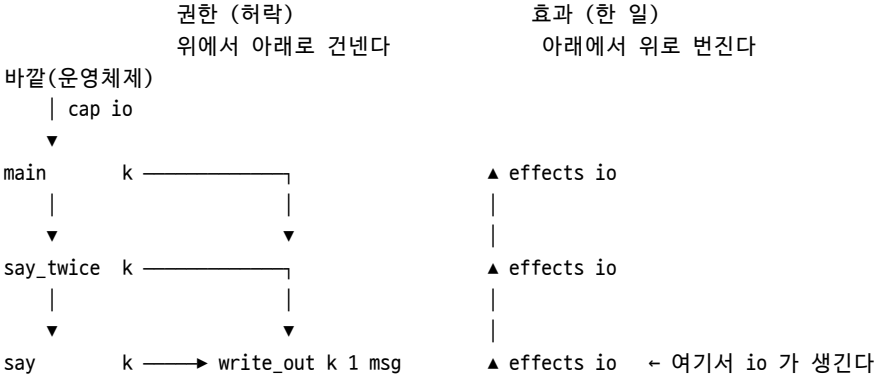
examples/ch16/chain.low

```
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
```

실행 결과

```
$ lowentc --run main chain.low
hi
hi
main() = 0
```

사슬은 이렇게 생긴다 — 바깥 → `main` 의 `k` → `say_twice` 의 `k` → `say` 의 `k` → `write_out`. 효과와 함께 그리면 두 화살표가 반대로 흐른다.



권한은 **위에서** 건네받아야 쓸 수 있고, 효과는 **아래에서** 생겨 부르는 쪽 머리마다 적힌다. 그래서 어느 op 의 머리만 보면 «이 op 이 무엇을 할 수 있고(효과), 누가 그것을 허락했는지(권한)» 가 다 보인다. 권한을 받지 않은 op 은 `say` 를 부를 수 없다. 넘길 것이 없기 때문이다. 그래서 "이 프로그램이 파일을 건드릴 수 있는가" 를 **시작점만 보고** 답할 수 있다. 깊은 곳의 어떤 op 이 몰래 파일을 여는 일은 있을 수 없다. 권한을 받지 않았다면 열 수 없고, 받았다면 그 사슬이 시작점까지 이어져 보인다.

> **문. 권한을 인자로 넘기면 실행할 때 인자가 하나씩 늘어 느려지지 않는가?**
>
> **답.** 늘지 않는다. 권한은 **번역 시점의 표시**이고 실행 중의 값이 아니다. 권한을 받는 op 과 받지 않는 op 은 C 경계에서 같은 서명을 갖는다. 권한은 "이 op 을 부를 자격이 있는가" 를 번역할 때 묻고, 실행할 때는 물을 것이 남지 않는다. 경계를 지키는 값이 실행 비용으로 돌아오지 않는다.

16.3 효과와 권한은 짝이다

효과 줄은 op 이 **무엇을 하는지**, 권한 입력은 **누가 허락했는지**를 적는다. 같은 일을 두 쪽에서 적고 서로를 검사한다.

효과	허락하는 권한	없을 때
`io`	`cap io` · `file_system` · `net` · `tty` · `clock` · `random`	`E-EFFECT-NO-CAP`
`alloc`	`cap allocator`(또는 영역)	`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`	권한 없이 적는다	—

표 16.2 — 효과와 그것을 허락하는 권한

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
```

실행 결과

```
$ lowentc --run main pairs.low
hi
main() = 0
```

`main` 이 두 권한을 받는다. `cap io` 가 `effects io` 를, `cap allocator` 가 `effects alloc` 을 허락한다. `alloc_bytes al capacity 16` 은 할당기에서 16 바이트를 청하고, 얻지 못할 수 있으므로 `option` 을 준다.

짝이 어긋나는 모양은 셋이다. 효과를 적었는데 허락하는 권한이 없으면 거절된다.

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
```

실행 결과

```
$ 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
```

할당도 같다. 메모리를 쓰겠다고 적고 그 자리를 어디서 받는지 적지 않았다.

examples/ch16/alloc_nocap.low

```
module alloc_nocap .
rem expect: E-ALLOC-NOCAP

proc scratch output u8 . effects alloc .
do
  return 1 .
end
```

실행 결과

```
$ 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)
```

권한을 받아 두기만 하고 쓰는 자리에 건네지 않아도 거절된다. 권한을 요구하는 내장 op 은 그 권한을 **첫 피연산자로** 적는다.

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
```

실행 결과

```
$ 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
```

이름 없이 권한에 닿는 길은 없다. 숨은 표준출력이 없다는 뜻이다.

16.4 있음이 아니라 종류다

권한 값을 하나 가지고 있다고 모든 문이 열리지는 않는다.

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
```

실행 결과

```
$ 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` 는 시계를 읽는 원시 연산이다. `cap io` 를 건넸으므로 종류가 틀렸다고 거절된다. 권한은 **무엇인가**로 인가된다.

> **흔한 오해. `cap clock` 으로 시각을 읽으면 `io` 효과를 적어야 한다**
>
> 표준 라이브러리의 `clock.now_ms` 는 `cap clock` 을 받지만 `effects none` 이다. 시계는 결정성을 깨므로 권한이 필요하지만, 바깥에 흔적을 남기지는 않는다. 권한과 효과가 늘 일대일은 아니다. 짝표의 `io` 줄은 "`io` 효과를 내려면 이 가운데 하나를 받아야 한다" 는 방향의 규칙이다.

권한의 **이름**은 저자가 지을 수 있다. `input logger cap audit .` 처럼 자기 권한을 만들어 매개변수로 받으면, 그 권한을 받은 op 만 부를 수 있는 op 을 만들 수 있다. 처리기가 뜻을 정해 둔 것은 위 표의 종류들이고, 그 종류의 일을 하려면 그 종류를 건네야 한다.

16.5 시작점이 받는 것

프로그램이 시작하는 op 은 **권한만** 받는다. 자료를 받지 않는다. 건네줄 사람이 없기 때문이다.

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
```

실행 결과

```
$ 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)
```

C 의 `argv` 같은 마법의 인자는 없다. 프로그램 인자도 `cap args` 로 받는 권한이다. 시작점이 받을 수 있는 권한은 닫힌 목록이다 — `args·env·io·allocator·heap·file_system·net·tty·clock·random·atomic`. 그 밖의 권한(`mmio·machine·c`)은 실행하는 쪽이 건넬 수 있는 것이 아니라서 시작점이 요구하면 `E-ENTRY-CAP` 으로 거절된다. 그런 권한은 그것을 줄 자격이 있는 자리에서 만들어져 인자로 흘러야 한다(29·30장).

> **실제 사례. 의존성이 네트워크에 닿는지 알고 싶을 때**
>
> 남이 만든 라이브러리를 들여올 때 가장 불안한 것은 그 코드가 무엇에 닿는지다. Lowent 에서는 그 라이브러리의 `export` 된 op 머리를 훑어 `cap net` 을 받는 op 이 있는지 보면 된다(C 로 빠져나가는 `cap c` 와 기계 명령의 `cap machine` 도 함께 본다 — 그 안은 처리기가 보지 못한다). 없다면 그 라이브러리는 연결을 열 수 없다. 있다면 그 op 을 부르는 자리에 내가 `cap net` 을 건네야 하므로, 그 자리가 곧 감사할 자리다. 공급망 공격의 흔한 모양 — 들여온 코드가 몰래 바깥에 연결한다 — 이 문법으로 드러난다.

환경 변수와 운영체제의 난수도 권한으로 받는다.

examples/ch16/envrandom.low

```
module envrandom .
rem run: main
```

```
use random .

rem 환경 변수는 cap env, 운영체제의 엔트로피는 cap random — 둘 다 받아야 닿는다
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 여덟 바이트를 운영체제의 난수로 채운다 — 값은 매번 다르므로 돌려주지 않는다
  let n u64 be random.bytes r buf .
  rem 환경 변수가 없으면 none — 값이 아니라 없음을 돌려준다
  let v option slice u8 be env_get e "LOWENT_MANUAL_DEMO" .
  guard is_some v . else return narrow u8 n .
  return 99 .
end
```

실행 결과

```
$ lowentc --run main envrandom.low
main() = 8
```

`env_get e "…"` 은 `cap env` 가, `random.bytes r buf` 는 `cap random` 이 있어야 부를 수 있다. 환경 변수는 프로그램 밖에서 조용히 동작을 바꾸는 통로이고, 난수는 같은 입력에도 답을 바꾸는 통로다. 둘 다 머리에 보여야 "이 프로그램의 답은 입력만으로 정해지는가" 를 머리만 보고 판단할 수 있다. 변수가 없으면 `env_get` 은 `none` 이고, 예제는 채운 바이트 수 8 을 돌려준다. 시험에서 되풀이할 수 있는 난수가 필요하면 권한이 필요 없는 `rng_next` (씨앗에서 다음 수)를 쓴다.

16.6 흔한 실수

반례. 권한을 받았으니 `fn` 도 출력할 수 있다고 생각한다

examples/ch16/mistake_fncap.low

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

rem ✘ 권한을 받았다고 `fn` 이 출력할 수 있게 되지는 않는다
fn say input out cap io . output u64 .
do
  return write_out out 1 "hi\n" .
end
```

실행 결과

```
$ 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
```

권한은 "누가 허락했는가" 이고 효과는 "무엇을 하는가" 다. 둘은 서로를 대신하지 않는다. `cap io` 를 받았어도 출력은 `io` 효과이고, 순수한 `fn` 은 효과를 낼 수 없으므로 `E-EFFECT-CALC` 다. `proc say … effects io .` 로 바꾼다. 허락(권한)과 행동(효과)을 모두 머리에 적어야 두 쪽이 서로를 검사한다.

반례. 받은 권한을 지역 이름에 옮겨 담는다

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 ✘ 권한을 지역 이름에 옮겨 담았다 --- 받은 이름을 그대로 건넨다
  let k cap io be out .
  return write_out k 1 "hi\n" .
end
```

실행 결과

```
$ 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)
```

권한은 사슬을 따라 **받은 이름 그대로** 넘긴다. 지역 이름에 옮겨 담으면 권한이 어디서 왔는지 사슬이 끊겨 보이므로 `E-CAP-LOCAL` 로 거절한다. 진단은 옮겨 담은 지역 이름과 그 종류를 짚어 원인을 말한다. `write_out out 1 …` 처럼 매개변수 이름을 곧바로 적는다.

반례. 다른 종류의 권한으로 표준출력에 쓴다

examples/ch16/mistake_fscapout.low

```
module mistake_fscapout .
rem expect: E-CAP-KIND

rem ✘ 파일 권한으로 표준출력에 쓰려 한다
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
```

실행 결과

```
$ 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` 은 파일과 디렉터리를 여는 권한이지 표준출력의 권한이 아니다. 권한은 **있음이 아니라 종류**로 인가되므로 `E-CAP-KIND` 로 거절된다 — 진단이 필요한 종류와 건넨 종류를 나란히 적는다. 고치는 길은 시작점이 `input out cap io .` 를 받아 그 이름을 건네는 것이다.

반례. 권한 자리에 수를 넘긴다

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 ✘ 권한을 하나도 받지 않았는데, 권한 자리에 수 0 을 넘겨 출력한다
proc start output u8 .
do
  let n u64 be say 0 .
  return 0 .
end
```

실행 결과

```
$ 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` 는 권한을 하나도 받지 않았다. 그러니 출력할 수 없어야 한다. `say` 의 `cap io` 자리에 수 `0` 을 넘기는 것이 그 규칙을 피해 가는 길처럼 보이지만, `E-CAP-FORGE` 로 거절된다. 권한 자리에 올 수 있는 것은 **건네받은 이름**(`input … cap …`)이나 액터의 권한 칸뿐이다. 이것이 막히지 않으면 "시작점만 보면 닿을 수 있는 바깥을 안다" 는 이 장의 약속이 한 줄로 무너진다. `--run` 이 쓰는 자리표 `0` 은 도구의 입구에서만 뜻이 있다(31장).

흔한 오해. 나중에 쓸지도 모르니 권한과 효과를 미리 받아 두면 편하다

examples/ch16/capjustincase.low

```
module capjustincase .
rem expect: W-EFFECT-OVER

rem 나중에 쓸지도 몰라 권한과 효과를 미리 적어 두었다
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
```

실행 결과

```
$ 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 는 곱셈만 하는데 cap io 와 effects io 를 적었다. 그러면 이 op 을 부르는 모든 쪽이 cap io 를 구해 건네야 하고, 순수한 계산에서는 부를 수조차 없다. 도구는 W-EFFECT-OVER 로 알린다. 권한은 최소한으로 받는다 — 받은 권한의 목록이 곧 "이 op 이 닿을 수 있는 바깥" 이라서, 넉넉하게 받으면 그 목록이 거짓말을 한다.

16.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`input out cap io .`	표준 입출력의 권한을 받는다	주변 권한이 없다 — 힘은 인자로 건네받는다
`input fs cap file_system . • cap net • cap clock …`	종류마다 여는 일이 다르다	최소 권한 — 머리가 닿을 수 있는 바깥을 말한다
`say k msg`	받은 권한 이름을 그대로 넘긴다	사슬이 시작점까지 이어져 보인다
`write_out out 1 "…"`	권한을 쓰는 내장 op 은 권한을 첫 피연산자로	이름 없이 권한에 닿는 길이 없다
`effects io + cap io`	효과와 그것을 허락하는 권한의 짝	없으면 E-EFFECT-NO-CAP
`effects alloc + cap allocator`	고정 창 할당의 짝	없으면 E-ALLOC-NOCAP
`proc main input … cap … . output u8 .`	시작점은 권한만 받는다	건네줄 사람이 없으니 자료는 받지 않는다
`input logger cap audit .`	저자가 이름 지은 권한	그 권한을 받은 op 만 부를 수 있게 한다
`env_get e "…" • random.bytes r buf`	환경 변수 · 운영체제 난수 — cap env · cap random	답을 바꾸는 바깥 통로가 머리에 보인다

표 16.3 — 권한의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

권한은 input <이름> cap <종류> . 로 건네받고, 받은 이름을 그대로 넘긴다. 시작점만 보면 프로그램이 닿을 수 있는 바깥을 안다. 효과와 권한은 짝이라 효과를 적으면 허락하는 권한을 받아야 하고, 권한을 쓰는 내장 op 은 권한을 첫 피연산자로 적는다. 권한은 종류로 인가되며 실행 비용이 없다. 시작점은 닫힌 목록의 권한만 받는다.

17 실패를 설계하기

이 장의 차례

17.1 경계와 안쪽 196
17.2 오류 열거형을 나누는 기준 197
17.3 층을 올라가며 198
17.4 정보를 버려도 되는 자리 199
17.5 `panic` 을 써도 되는 자리 199
17.6 흔한 실수 200
17.7 이 장의 문법 한눈에 204

먼저 알아야 할 것

11장 답을 담는 타입 · 실패를 말하는 세 길
14장 계약 · `requires` 는 부르는 쪽, `errors` 는 op 의 책임
16장 권한 · 시작점만 보면 바깥과 닿는 자리를 안다

돌아보기

11장은 `result`·`option`·`panic` 을 가르는 물음을 무엇이라 했는가?

> **답.** “부르는 쪽이 무엇을 할 수 있는가” 였다. 다른 길을 시도할 수 있으면 `result`, 그냥 없는 것이면 `option`, 부르는 쪽이 이미 약속을 어긴 것이라 고칠 수 없으면 멈춘다. 이 장은 그 물음을 op 하나가 아니라 **프로그램의 층** 전체에 적용한다.

이 장의 필요성과 맥락

문법을 다 알아도 실패를 어디서 무엇으로 말할지는 저절로 정해지지 않는다. 같은 “잘못된 포트 번호”가 설정 파일을 읽는 층에서는 `result` 여야 하고, 이미 검사를 마친 안쪽 층에서는 계약이어야 한다. 이 선택을 틀리면 검사가 여러 겹으로 되풀이되거나, 반대로 아무 층도 검사하지 않는다. 제4부의 마지막 장으로, 계약·오류·권한을 함께 써서 실패의 자리를 정하는 법을 정리한다.

이 장이 끝나면

바깥에서 온 값과 안쪽의 불변식을 가르고, 경계에서는 `result` 로, 안쪽에서는 계약으로 말하는 설계를 익힌다. 오류 열거형을 어떻게 나누고 `errors` 절에 무엇을 적을지, 층을 올라가며 `try` 로 넘길지 그 자리에서 다룰지 정하는 기준을 알게 된다. `option` 으로 정보를 버려도 되는 자리와 `panic` 을 써도 되는 자리도 보게 된다.

이 장에서 답할 질문

1 `errors not_digit .` 처럼 조건 없이 적은 오류와 `errors empty eq (len s) 0 .` 처럼 조건을 붙인 오류는 무엇이 다른가?

17.1 경계와 안쪽

프로그램에는 **바깥에서 값이 들어오는 자리**가 있다. 프로그램 인자, 파일 내용, 네트워크에서 온 바이트, 사용자의 입력이다. 그 값에는 아무 약속도 없다. 반대로 한 번 검사를 통과한 값이 흘러 다니는 **안쪽**이 있다.

설계의 첫 원칙은 이 둘을 가르는 것이다.

자리	값의 성격	말하는 법
경계(파싱·입력)	무엇이든 올 수 있다	`result` 와 `errors`
안쪽(계산)	이미 검사를 통과했다	`requires`·`range`·`newtype`
어느 층도 다룰 수 없음	있어서는 안 되는 상태	`panic`

표 17.1 — 실패를 말하는 자리

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 바깥에서 온 바이트 — 무엇이든 올 수 있으므로 실패는 값이다
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 이 층은 «왜» 가 필요 없다 — 없으면 기본값을 쓴다
fn port_or_default input s slice u8 . output u16 .
do
  let r result u16 parse_error be parse_u16 s .
```

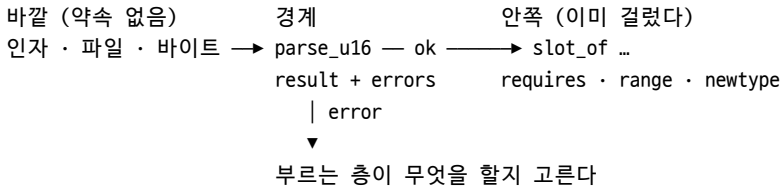
```
  guard not (is_error r) . else return 8080 .
  return ok_value r .
end
```

실행 결과

```
$ 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` 은 경계다. 바이트는 비어 있을 수도, 숫자가 아닐 수도, 너무 클 수도 있다. 셋 모두 **고칠 수 있는 실패**이므로 `result` 로 돌려준다. 계약으로 "숫자만 주시오" 라고 요구하는 것은 틀린 설계다 — 부르는 쪽은 바이트를 받자마자 이 op 에 넘기므로, 요구를 지킬 방법이 없다.

값이 어디서 와서 어디로 가는지 그리면 이렇다.



> **문. `errors not_digit .` 처럼 조건 없이 적은 오류와 `errors empty eq (len s) 0 .` 처럼 조건을 붙인 오류는 무엇이 다른가?**

> **답.** 조건을 붙이면 "이 조건일 때 **정확히** 이 오류가 난다" 는 나가는 쪽의 계약이 된다. 조건이 참인데 정상으로 돌아오면 계약 위반이다. 부르는 쪽은 그 조건만 피하면 그 오류를 다루지 않아도 된다고 믿을 수 있다. 조건을 계약의 식으로 적을 수 없는 오류(모든 바이트가 숫자인가)는 조건 없이 적어 "날 수 있다" 만 약속한다. 적을 수 있는 조건은 적는 편이 부르는 쪽에 더 많은 것을 알려 준다.

17.2 오류 열거형을 나누는 기준

오류 갈래는 **부르는 쪽이 다르게 행동할 경우**마다 하나씩 둔다. `empty·not_digit·too_big` 을 가른 것은 화면에 다른 안내를 보여 줄 수 있기 때문이다. 부르는 쪽이 모두 같은 일을 한다면 갈래를 합친다. 갈래가 많을수록 `match` 로 다루는 쪽의 짐이 는다.

반대로 한 op 이 서로 관계없는 층의 오류를 모두 담는 거대한 열거형은 피한다. 파일 오류와 파싱 오류와 설정 오류가 한 열거형에 섞이면, 머리의 `errors` 절이 그 op 이 실제로 낼 수 있는 것보다 넓은 그림을 그리게 되고, 일어날 수 없는 오류 선언은 번역이 거절한다(14장).

17.3 층을 올라가며

실패를 받은 층은 둘 중 하나를 한다. **넘기거나**, **그 자리에서 다루거나**.

examples/ch17/layers.low

```
module layers .
rem run: main

enum config_error do
  bad_port .
  port_zero .
end

rem 안쪽 불변식 — 부르는 쪽이 이미 걸렀어야 하는 것은 계약이다
fn slot_of input port u16 . input slots u64 . output u64 .
  requires gt slots 0 .
do
  return mod (widen u64 port) slots .
end

rem 바깥 입력 — 틀릴 수 있는 것은 result 다
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
```

실행 결과

```
$ lowentc --run main layers.low
slot ok
main() = 0
```

- `slot_of` 는 안쪽이다. 슬롯 수가 0 이면 나머지 연산이 멈추지만, 그것은 부르는 쪽의 잘못이므로 `requires gt slots 0 .` 으로 적었다.
- `check_port` 는 경계에 가깝다. 0 번 포트는 설정이 틀린 것이고 고칠 수 있으므로 `result` 다.
- `pick_slot` 은 `try` 로 실패를 위로 넘긴다. 자기도 같은 오류를 `errors` 에 적었다. 성공한 뒤에는 `p` 가 0 이 아님을 알므로 안쪽 op 을 부른다.
- `main` 은 가장 바깥 층이다. 더 넘길 곳이 없으므로 실패를 **다룬다** — 메시지를 쓰고 종료 코드 1 을 돌려준다.

`port` 가 0 일 때 실패가 올라가는 길을 그리면 이렇다. 내려갈 때는 부르고, 올라올 때는 넘기거나 다룬다.

```
main ──────────────────────  다룬다: "bad port" 를 쓰고 1 을 돌려준다
 │  ▲ error port_zero
 ▼  │
pick_slot ─────────────────  넘긴다: try 가 그대로 위로
 │  ▲ error port_zero      │
 ▼  │                      ▼ 성공한 뒤에만
check_port                 slot_of
(경계: result)             (안쪽: requires gt slots 0)
```

규칙으로 정리하면, **넘길 수 있는 가장 가까운 층이 다룰 수 있으면 넘기지 말고, 다룰 수 없으면 넘긴다.** `try` 는 넘기는 비용을 한 낱말로 줄였지만, 넘긴다는 사실은 부르는 쪽의 `errors` 절에 남는다.

> **흔한 오해. 모든 op 이 `result` 를 돌려주면 가장 안전하다**
>
> 안쪽까지 `result` 를 돌려주면 모든 호출에 `try` 가 붙고, 모든 머리에 `errors` 가 붙는다. 그런데 안쪽에서는 그 실패가 **일어날 수 없다** — 경계에서 이미 걸렀기 때문이다. 일어날 수 없는 실패를 적으면 다루는 코드가 생기고, 그 코드는 영영 돌지 않는다. 안쪽의 불변식은 계약과 타입(`range`·`newtype`)으로 적는다. 그러면 검사가 경계에 한 번 남고, 안쪽에서는 사실로 쓰여 오히려 지워진다.

17.4 정보를 버려도 되는 자리

`port_or_default` 는 실패의 **이유**가 필요 없다. 무엇이 틀렸든 기본 포트 8080 을 쓴다. 이런 층에서는 `result` 를 묻고 버리거나, `try … else_none` 으로 `option` 으로 바꾸어 `value_or` 를 쓴다. 정보를 버리는 것은 **그 층이 결정할 일**이다. 아래 층이 미리 `option` 으로 버려 두면, 이유가 필요한 위 층이 되찾을 길이 없다. 그러니 버리는 것은 가능한 한 위에서 한다.

17.5 `panic` 을 써도 되는 자리

`panic` 은 복구할 수 없는 상태를 만났을 때 쓴다. 판단 기준은 “이 상태를 어느 층이 다룰 수 있는가” 다.

- 프로그램의 불변식이 깨졌다(내부 표가 손상되었다). 어느 층도 올바른 답을 낼 수 없으므로 멈춘다.
- 계속 가면 더 큰 피해가 난다(검증에 실패한 데이터를 기록하려 한다).

반대로 사용자의 입력이 틀린 것, 파일이 없는 것, 연결이 끊긴 것은 `panic` 할 일이 아니다. 어느 층인가 다룰 수 있다. 그리고 `panic` 은 효과이므로, 쓰는 순간 그 op 과 부르는 모든 op 의 머리에 `effects panic` 이 번진다 (15장). 그 비용이 `panic` 을 아껴 쓰게 만든다.

실제 사례. HTTP 요청 파서의 알맹이는 거절이다

표준 라이브러리의 `http` 모듈은 HTTP/1.1 요청을 읽는다. 그 문서가 스스로를 소개하는 말이 "알맹이는 **거절**이다" 이다. 요청 밀반입(request smuggling) 같은 공격은 서버와 프록시가 애매한 요청을 **서로 다르게 받아들이는** 자리에서 산다. 그래서 파서는 애매한 입력을 최대한 받아 주지 않고 이름 붙은 오류로 돌려준다. 경계의 op 이 `result` 로 무엇을 거절하는지 분명하게 말하면, 안쪽은 받아들인 요청의 모양을 믿을 수 있다(36장).

17.6 흔한 실수

반례. 경계에서 틀린 입력에 `panic` 한다

examples/ch17/mistake_panicinput.low

```
module mistake_panicinput .
rem run: parse_or_die [52,50]
rem trap: parse_or_die [52,120]

rem ✘ 사용자가 잘못 친 한 글자에 프로그램 전체가 멈춘다
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
```

실행 결과

```
$ 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)
```

사용자가 `4x` 를 쳤을 뿐인데 프로그램 전체가 멈춘다. 틀린 입력은 **흔히 일어나는 일**이고 어느 층인가 다룰 수 있다 — 다시 묻거나, 기본값을 쓰거나, 안내를 보여 줄 수 있다. `panic` 은 그 선택지를 모두 없앤다. 게다가 `effects panic` 이 부르는 쪽 모두로 번진다. 이 장 첫머리의 `parse_u16` 처럼 실패를 `result` 로 돌려주고, 무엇을 할지는 부르는 층이 정하게 한다.

반례. 아래 층에서 이유를 미리 버린다

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 ✘ 아래 층이 미리 이유를 버렸다
fn digit_or_none input s slice u8 . output option u8 .
do
  return try (parse_digit s) else_none .
end

rem 위 층은 빈 입력과 숫자 아닌 입력을 다르게 안내하고 싶지만 둘 다 0 만 본다
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
```

실행 결과

```
$ lowentc --run explain mistake_dropwhy.low []
explain([]) = 0
  arg0 (written) = []
$ lowentc --run explain mistake_dropwhy.low [120]
explain([120]) = 0
  arg0 (written) = [120]
```

`digit_or_none` 이 `else_none` 으로 오류를 `option` 으로 바꾸는 순간 “왜” 가 사라진다. 위 층의 `explain` 은 빈 입력과 숫자 아닌 입력에 다른 안내를 하고 싶지만 둘 다 0 만 본다. 되찾을 길이 없다. 이유는 끝까지 들고 올라와, 버려도 되는지 아는 층에서 버린다.

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 이유를 끝까지 들고 올라와, 필요한 층에서 가른다
fn explain input s slice u8 . output u8 .
do
  match parse_digit s do
    case ok v . return 9 .
    case error e . do
      rem 오류 값에 이름을 묶고, 그 열거형을 한 번 더 가른다
      match e do
        case empty . return 1 .
        case not_digit . return 2 .
      end
    end
  end
end
```

실행 결과

```
$ 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]
```

오류 갈래는 `case error <갈래이름>` 으로 곧바로 가른다.

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 `case error <갈래>` 의 이름이 선언된 갈래면 그 갈래만 받는다
  match parse_digit s do
    case ok v . return 9 .
    case error empty . return 1 .
    case error not_digit . return 2 .
  end
end
```

실행 결과

```
$ 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]
```

`case error` 뒤의 이름이 **선언된 갈래**면 그 갈래만 받는다. 선언된 갈래가 아니면 오류 값 전체를 그 이름에 묶는다(`case error e`) — 이름 하나로 두 가지 뜻을 쓰는 셈이지만, 가르는 규칙은 `match` 의 다른 자리와 같다: 맨 이름이 갈래면 갈래다(7장). 갈래마다 적으면 망라 검사가 갈래 하나가 빠진 것을 잡는다. 이름으로 묶으면 그 하나가 모든 오류를 받으므로 잡을 것이 없다.

반례. `result` 를 문장으로 불러 버린다

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 ✘ 검사의 결과를 문장으로 버렸다 --- 포트가 0 인데 "started" 가 나온다
  check_port 0 .
  let n u64 be write_out out 1 "started\n" .
  return 0 .
end
```

실행 결과

```
$ 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 .` 은 오류를 돌려주었지만 아무도 받지 않았다. 포트가 0 인데 "started" 가 나오고 종료 코드도 0 이 된다 — 실패를 값으로 돌려준다는 설계는 부르는 쪽이 그 값을 **볼 때만** 지켜진다. 그래서 도구가 `W-RESULT-DISCARD` 로 알린다. `result` 를 돌려주는 op 은 `let` 으로 받아 묻거나 `try` 로 넘기거나 `match` 로 가른다. 이름에 담기만 하고 한 번도 읽지 않는 것도 같은 알림을 받는다 — 실패가 사라지는 것은 마찬가지이기 때문이다. 실패로 할 일이 정말 없는 자리(오류 경로에서 닫고 나가는 자리)에서는 `drop <이름> .` 으로 «일부러 넘긴다» 를 적는다. 잘못은 무시하는 것이 아니라 **말없이** 무시하는 것이다.

17.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
경계의 op `output result t e . + errors`	바깥 값의 실패를 값으로 돌려준다	부르는 층이 무엇을 할지 고른다
안쪽의 op `requires` · `range` · `newtype`	이미 걸러진 값의 불변식	검사가 경계에 한 번 남고 안쪽에서는 지워진다
`enum parse_error do empty . not_digit . end`	부르는 쪽이 다르게 행동할 경우마다 갈래 하나	갈래가 많을수록 다루는 짐이 는다
`let v u16 be try check_port port .`	다룰 수 없으면 위로 넘긴다	넘긴 사실이 `errors` 절에 남는다
`case error e . do match e do … end end`	오류 값을 묶고 갈래를 한 번 더 가른다	`case error <이름>` 의 이름이 선언된 갈래가 아니면 새 묶음이다
`try … else_none` · `value_or`	이유를 버린다	버리는 것은 가능한 한 위 층에서
`proc … effects panic . + panic "…"`	어느 층도 다룰 수 없는 상태에서 멈춘다	틀린 입력·없는 파일에는 쓰지 않는다
시작점 `output u8 .`	가장 바깥 층 — 실패를 다루고 종료 코드로 알린다	더 넘길 곳이 없다

표 17.2 — 실패를 설계하는 모양 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

바깥에서 값이 들어오는 경계에서는 실패를 `result` 와 `errors` 로 말하고, 검사를 통과한 안쪽에서는 계약과 타입으로 불변식을 적는다. 오류 갈래는 부르는 쪽이 다르게 행동할 경우마다 둔다. 실패를 받은 층은 다룰 수 있으면 다루고 없으면 `try` 로 넘긴다. 정보를 버리는 일은 가능한 한 위 층에서 하고, `panic` 은 어느 층도 다룰 수 없는 상태에만 쓴다.

제5부 — 메모리

18 영역 — 값이 사는 곳과 한꺼번에 걷히는 메모리

이 장의 차례

18.1 값이 사는 세 곳 207
18.2 영역을 열고 자리를 얻는다 207
18.3 영역을 건네받는다 209
18.4 영역 밖으로 들고 나갈 수 없다 210
18.5 뿌리마다 커서는 하나다 211
18.6 자라는 뿌리 212
18.7 영역 위의 스택 213
18.8 흔한 실수 214
18.9 이 장의 문법 한눈에 218

먼저 알아야 할 것

12장 빌리기 · 빌림은 빌려준 값보다 오래 살 수 없다
16장 권한 · `effects alloc` 은 `cap allocator` 와 짝이다

돌아보기

12장에서 op 안의 지역을 가리키는 참조를 돌려주면 무엇으로 거절되었는가? 그리고 값을 op 밖으로 내보내야 할 때 권한 길은 무엇이었는가?

> **답.** `E-ESCAPE` 로 거절되었다. 지역은 op 이 끝나면 사라지므로 그 참조는 없는 값을 가리킨다. 값을 내보내려면 참조가 아니라 값 자체를 돌려주거나, 호출자가 넘긴 저장소에 담는다. 이 장은 그 "저장소"가 어디서 오는지를 다룬다.

이 장의 필요성과 맥락

쓰레기 수거기가 없는 언어는 메모리를 언제 돌려줄지 누군가 정해야 한다. C 는 사람에게 맡겼고, 사람은 두 번 돌려주거나 잊는다. Rust 는 값마다 소유자를 두어 소유자가 사라질 때 돌려준다. Lowent 는 먼저 더 굵은 단위를 둔다 — **영역**이다. 함께 태어나고 함께 죽는 값들을 한 영역에 담고, 영역이 끝날 때 한꺼번에 걷는다. 파서의 임시 노드, 요청 하나를 처리하는 동안의 버퍼처럼 시스템 프로그램의 메모리 대부분이 이 모양이다. 그래서 제5부는 개별 소유 (19장)보다 영역을 먼저 다룬다.

이 장이 끝나면

값이 사는 세 곳(지역·정적·얻은 것)과 얻는 뿌리 둘(고정 창·힙)을 알게 된다. `region <이름> <종류> do … end` 블록으로 영역을 열고 `alloc_bytes` 로 자리를 얻는 법, 영역을 매개변수로 건네는 법을 익힌다. 영역에서 얻은 값을 밖으로 들고 나가거나 안쪽 영역이 열린 채 바깥에서 깎으면 왜 거절되는지, 운영체제가 없는 기계에서 힙이 왜 막히는지도 보게 된다.

이 장에서 답할 질문

1 `make point do … end` 나 `some 7` 이 만드는 값은 어디에 사는가? 할당인가?

18.1 값이 사는 세 곳

저장되는 값은 셋 가운데 한 곳에 산다.

갈래	설명
지역	op 안에서 나고 op 이 끝나면 사라진다. 가장 흔하다
정적	프로그램이 사는 동안 계속 있다
얻은 것	뿌리에서 얻는다. 영역이나 소유가 언제 돌려줄지 정한다

표 18.1 — 값이 사는 곳

흔히 스택과 힙이라 부르는 것이 지역과 얻은 것에 해당한다. 이름을 달리 쓰는 것은 이 언어에서 그것이 기계의 구조가 아니라 **값의 성질**이기 때문이다. 그리고 어느 곳에 사는지는 소스에 적혀 있다. 처리기가 몰래 옮기지 않는다.

얻는 뿌리는 둘이고 서로 따로 깎이고 따로 되감긴다.

뿌리	무엇으로 깎나	효과	성질
고정 창	`cap allocator · heap` 이 아닌 영역	`alloc`	**자라지 않는다**. 다 쓰면 `none` 이다
힙	`cap heap · region <이름> heap`	`heap`	**자란다**. 운영체제가 있는 기계에만 있다

표 18.2 — 두 뿌리

고정 창은 운영체제가 없는 기계에서도 된다. 그런 기계에서는 링커가 정한 두 경계 사이가 창이다 (20장).

> **문.** `make point do … end` 나 `some 7` 이 만드는 값은 어디에 사는가? 할당인가?

> **답.** 처리기의 **유한 풀**에 산다. 그 op 의 암묵적인 틀(frame)이라서 효과가 없고 권한도 필요 없다. 풀은 반복을 한 바퀴 돌 때마다 되감기고, 동시에 살아 있는 값이 풀의 크기를 넘으면 그 자리에서 멈춘다. 크기는 기계가 정하고 짓는 사람이 조절할 수 있다. 작은 묶음 값을 만들 때마다 `alloc` 을 적지 않아도 되는 이유다.

18.2 영역을 열고 자리를 얻는다

영역은 `region <이름> <종류> do … end` 로 연다. 블록 안에서 `alloc_bytes <영역> capacity <n>` 이 그 영역에서 `n` 바이트를 청한다.

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
```

실행 결과

```
$ lowentc --run main scratch.low
main() = 70
```

- `region work arena do … end` 가 영역을 연다. `arena` 는 "앞에서부터 깎고 한꺼번에 되돌린다" 는 종류다.
- `alloc_bytes work capacity n` 은 `option mut slice u8` 을 준다. 자리가 모자라면 `none` 이다. 메모리가 모자란 것도 **값**이다.
- `fill_count` 의 머리에 `effects alloc` 이 있다. 자리를 얻는 것은 효과다. 순수한 `fn` 은 영역을 쓸 수 없다.
- 블록을 나가는 **모든 길** — 끝에 닿기, `return`, 감싼 반복의 `break`·`continue` — 에서 영역은 되감긴다. `return s .` 로 떠나도 버퍼는 걷힌다. 개별로 돌려주는 코드가 없다.

`main` 은 `cap allocator` 를 받는다. `fill_count` 의 `alloc` 효과가 번져 오므로, 그 효과를 허락할 권한이 필요하다 (16장). 영역 블록은 그 블록을 연 op 안에서 자리를 얻을 자격이 되지만, 효과는 부르는 쪽으로 번진다.

영역의 종류는 닫힌 여덟이다 — `stack`·`frame`·`arena`·`static`·`heap`·`mmap`·`disk`·`device`. 그 밖의 낱말은 거절된다.

examples/ch18/kinds.low

```
module kinds .
rem run: carve_stack
rem run: carve_static

rem 종류 낱말은 이 영역이 무엇을 위한 자리인지 말한다. 이 판에서는 heap 만 자라는 뿌리에
서, 나머지는 고정 창에서 깎는다
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
```

실행 결과

```
$ lowentc --run carve_stack kinds.low
carve_stack() = 32
$ lowentc --run carve_static kinds.low
carve_static() = 64
```

종류 낱말은 **이 영역이 무엇을 위한 자리인지**를 코드에 남긴다. 이 판에서 실제로 다르게 동작하는 것은 `heap` 하나다 — 자라는 뿌리에서 깎는다. 나머지 일곱은 모두 고정 창에서 깎고 되감으므로 `carve_stack` 과 `carve_static` 은 같은 방식으로 32 와 64 를 얻는다. 그래도 종류를 적고 그 목록을 닫아 둔 까닭은 둘이다. 기계에 맞춰 실체를 달리할 때 코드를 고치지 않으려는 것, 그리고 아무 낱말이나 쓸 수 있으면 그 낱말이 아무것도 말해 주지 못하기 때문이다. `region t arena` 를 읽은 사람은 "앞에서 깎고 한꺼번에 돌려준다" 를 안다.

```
고정 창:
    [ a 16 ][ b 32 ][ c 8 ][ ····· 비어 있음 ····· ]
                            ▲ 커서 --- 다음 alloc_bytes 가 여기서부터 깎는다
end 에 닿으면:
    [ ····· ]
    ▲ 커서가 영역을 연 자리로 되감긴다 --- a · b · c 가 한꺼번에 사라진다
```

값마다 돌려주는 코드가 없는데도 새는 것이 없는 까닭이 이 그림이다. 돌려주는 일은 커서 하나를 되돌리는 것이다.

18.3 영역을 건네받는다

영역은 인자로도 건넨다. `input <이름> region <타입> .` 으로 받고, 타입은 그 영역을 부르는 이름이다.

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
```

실행 결과

```
$ lowentc --run main param.low
main() = 30
```

`sum_squares` 는 스스로 영역을 열지 않고, 부르는 쪽이 연 영역 `r` 을 받아 그 안에서 깎는다. 그래서 받은 버퍼의 수명은 부르는 쪽의 영역이 정한다. 영역에서 얻은 것은 영역보다 오래 살 수 없고, 영역이 끝나면 거기서 얻은 것도 끝난다.

18.4 영역 밖으로 들고 나갈 수 없다

영역에서 얻은 바이트를 영역 밖의 이름에 담으면 거절된다.

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
```

실행 결과

```
$ 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
```

영역이 `end` 에서 닫히면 그 바이트는 되감긴다. 밖에 남은 `keep` 은 없는 것을 가리키게 된다. 들고 나가는 자리는 `return`, 영역 밖 이름에 대입하기, 영역 밖 이름의 칸이나 원소에 대입하기다. 반대로 정수·참거

짓처럼 영역의 바이트를 들지 않는 값 — `len buf` 나 합계 — 은 들고 나가도 된다. `fill_count` 가 합을 돌려준 것이 그 모양이다.

> **흔한 오해. 영역은 결국 수명을 사람이 관리하는 것이다**
>
> 수명을 **정하는** 것은 사람이지만, 그 수명을 **지키는** 것은 번역이다. 영역에서 얻은 값이 밖으로 새는 길, 닫힌 영역을 가리키는 길은 번역에서 막힌다. 사람이 하는 일은 "이 값들은 함께 죽는다" 를 블록으로 적는 것이고, 그 블록은 눈에 보인다. C 의 `free` 는 흩어져 있고 검사되지 않는다.

18.5 뿌리마다 커서는 하나다

같은 뿌리의 영역이 안쪽에 열려 있는 동안 바깥 영역의 이름으로 깎으면 거절된다.

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
```

실행 결과

```
$ 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
```

고정 창에는 깎는 자리(커서)가 하나뿐이다. 안쪽 영역이 끝날 때 커서를 되감으면 그 사이에 바깥 이름으로 받은 바이트까지 함께 되감긴다.

```
          ▼ outer 를 연 자리
                    ▼ inner 를 연 자리
          [ a ][ … ][ b ][ ✘ c ]
                    ▲ inner 이 끝나면 커서가 여기로 되감긴다 → c 도 함께 사라진다
a = outer 의 것 · b = inner 의 것 · c = inner 가 열린 동안 outer 이름으로 깎은 것 (✘ E-ALLOC-NESTED)
```

다른 뿌리 — 힙 영역 안에서 고정 창 권한으로 깎는 것 — 는 커서가 따로라서 상관없다.

같은 이유로 태스크로 띄우는 op 은 뿌리에서 곧장 얻지 못한다(`E-ALLOC-TASK`). 커서가 실행 흐름 사이에 나뉘지 않기 때문이다. 흐름 사이에서 할당기를 나누려면 커서를 원자적으로 옮기는 할당기가 필요하다(27장).

18.6 자라는 뿌리

`region <이름> heap` 은 자라는 뿌리에서 깎는다.

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
```

실행 결과

```
$ lowentc --run main heap.low
main() = 0
```

효과는 `alloc` 이 아니라 `heap` 이고, 짝이 되는 권한은 `cap heap` 이다. 힙 영역도 블록을 나갈 때 한꺼번에 걷힌다. 다만 이미 준 바이트를 옮기지 않는다.

운영체제가 없는 기계를 대상으로 지으면 힙은 어느 자리로도 청할 수 없다.

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
```

실행 결과

```
$ 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` 은 운영체제 없는 마이크로컨트롤러다. `heap` 효과, `cap heap` 입력, `region … heap` 블록이 모두 거절된다. 같은 기계에서 고정 창을 깎는 `alloc` 은 쓸 수 있다. 코드가 어느 뿌리를 딛는지가 머리에 적혀 있으므로, 어떤 라이브러리가 운영체제 없는 기계에서 도는지는 번역이 답한다.

18.7 영역 위의 스택

영역에서 얻는 것은 바이트만이 아니다. `stack_new <영역> capacity <n>` 은 그 영역에 원소 `n` 개짜리 스택을 만든다. 되풀이로 나무나 그래프를 훑을 때처럼 "나중에 넣은 것을 먼저 꺼내는" 일에 쓴다.

examples/ch18/stack.low

```
module stack_demo .
rem run: main

type scratch u64 .

rem 영역에서 스택 하나를 얻어 자릿수를 거꾸로 쌓는다 --- 나중에 넣은 것이 먼저 나온다
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
```

실행 결과

```
$ lowentc --run main stack.low
main() = 74
```

- `let work stack u64 be stack_new temp capacity 8 .` 이 스택을 만든다. 영역에서 자리를 얻으므로 `effects alloc` 이다.

- `push work x .` 는 넣는다.
- `while pop work into d . do … end` 는 꺼낼 것이 있는 동안 하나씩 꺼내 `d` 에 묶는다. 비면 되풀이가 끝난다 — 빈 스택에서 꺼내다 멈추는 길이 문법에 없다.

47 의 자릿수 7·4 를 넣으면 4·7 차례로 나와 74 가 된다. 스택도 영역과 함께 걷히므로 따로 돌려주는 코드가 없다.

18.8 흔한 실수

반례. 자리를 얻었는지 묻지 않고 `some_value` 로 꺼낸다

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 ✘ 자리를 얻었는지 묻지 않고 꺼낸다 --- 창이 모자라면 `none` 을 꺼내다 멈춘다
    let buf mut slice u8 be some_value (alloc_bytes work capacity n) .
    return len buf .
  end
  return 0 .
end
```

실행 결과

```
$ 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 바이트는 얻지만 백만 바이트는 고정 창에 들어가지 않는다. `alloc_bytes` 는 그때 `none` 을 주고, 묻지 않고 꺼낸 `some_value` 가 `E-VM-NONE` 으로 멈춘다. C 의 `malloc` 결과를 `NULL` 과 비교하지 않는 것과 같은 실수인데, Lowent 는 없는 자리를 쓰는 대신 꺼내는 자리에서 멈춘다. 이 장의 예제처럼 `guard is_some g . else return 0 .` 으로 먼저 묻는다. 메모리가 모자란 것도 다뤄야 할 **값**이다.

반례. 영역에서 얻은 버퍼를 돌려준다

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 ✘ 이 블록이 끝나면 걷힐 바이트를 돌려준다
    return some_value g .
```

```
  end
  return subslice "" 0 0 .
end
```

실행 결과

```
$ 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
```

`make_buf` 가 돌려준 바이트는 `end` 에서 되감기므로, 부르는 쪽이 받는 순간 다음 할당이 그 자리를 남에게 줄 수 있다. C 에서 지역 배열의 주소를 돌려주는 결함과 같은 모양이고 `E-REGION-ESCAPE` 로 거절된다. 진단의 말대로 블록을 바깥으로 옮긴다. 부르는 쪽이 영역을 열어 건네고, 받은 쪽이 그 안에서 깎는다.

examples/ch18/returnbuf_fixed.low

```
module returnbuf_fixed .
rem run: main

type scratch u64 .

rem 영역을 연 쪽이 아니라 받은 쪽이 깎는다 --- 버퍼의 수명은 부르는 쪽의 블록이 정한다
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
```

실행 결과

```
$ lowentc --run main returnbuf_fixed.low
main() = 5
```

반례. 영역을 쓰는 op 을 `fn` 으로 적는다

examples/ch18/mistake_fnregion.low

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

rem ✘ 한꺼번에 돌려주니 바깥에 남는 것이 없다고 보고 `fn` 으로 적었다
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
```

실행 결과

```
$ 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
```

블록이 끝나면 모두 되감기니 바깥에 남는 것이 없어 보인다. 그래도 자리를 얻는 것은 `alloc` 효과다. 창에 자리가 있는지에 따라 결과가 달라질 수 있고(none), 같은 창을 쓰는 다른 코드와 겹친다. 그래서 `E-EFFECT-CALC` 다. `proc … effects alloc .` 으로 적는다.

반례. 영역 밖에서 태어난 액터에게 영역의 바이트를 건넨다

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 ✘ 영역 밖에서 태어난 액터에게 영역의 바이트를 건넨다
      let c u64 be send a init (some_value g) .
    end
  end
  return 0 .
end
```

실행 결과

```
$ 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
```

액터 `a` 는 영역보다 먼저 태어났으니 영역이 닫힌 뒤에도 산다. 그 액터가 받은 슬라이스를 상태에 넣어 두는지는 번역할 때 알 수 없다. 넣어 둔다면 영역의 `end` 가 그 바이트를 다른 쪽에 준 뒤에도 옛 자리를 읽는다. 그래서 보수적으로 `E-ALLOC-OUTLIVES` 로 막는다. 액터도 영역 안에서 만들면 둘이 함께 사라진다.

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 액터도 영역 안에서 태어난다 — 둘이 함께 사라진다
      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
```

실행 결과

```
$ lowentc --run carve_inside outlives_fixed.low
carve_inside() = 10
```

흔한 오해. 반복 안에서 얻은 자리는 바퀴마다 돌려준다

examples/ch18/loop_region.low

```
module loop_region .
rem run: count_outer
rem run: count_inner

rem 영역이 반복 바깥에 있다 --- 바퀴마다 얻은 4096 바이트가 블록 끝까지 쌓인다
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 영역을 반복 안에서 연다 --- 바퀴가 끝날 때마다 되감긴다
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
```

실행 결과

```
$ lowentc --run count_outer loop_region.low
count_outer() = 16
$ lowentc --run count_inner loop_region.low
count_inner() = 100
```

영역은 **블록**이 끝날 때 되감긴다. 바퀴가 끝날 때가 아니다. `count_outer` 는 영역이 반복 바깥에 있어서 4096 바이트씩 쌓이고, 이 판의 기본 고정 창(65536 바이트)에서 열여섯 번째 뒤로는 `none` 을 받는다. `count_inner` 는 영역을 바퀴 안에서 열어 매번 되감기므로 백 번 모두 얻는다. 한 바퀴에서만 쓰는 버퍼는 영역을 반복 안에서 연다.

18.9 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`region work arena do … end`	영역을 연다 — 블록을 나가는 모든 길에서 한꺼번에 되감긴다	`free` 가 없다 — 수명이 곧 블록
`alloc_bytes work capacity n`	영역에서 `n` 바이트를 청한다 — `option mut slice u8`	모자람도 값이다
`effects alloc` · `effects heap`	고정 창에서 얻는다 · 자라는 힙에서 얻는다	어느 뿌리를 딛는지 머리에 보인다
`input al cap allocator .` · `cap heap`	시작점이 받는 할당 권한	효과를 허락하는 짝
`input temp region scratch .`	부르는 쪽이 연 영역을 받는다	버퍼의 수명을 부르는 쪽이 정한다

stack·frame·arena·static·heap·mmap·disk·device	닫힌 영역 종류 여덟	낱말이 뜻을 말하게 — 밖의 낱말은 E-REGION-KIND
영역 밖 이름에 담기 · 돌려주기	거절(E-REGION-ESCAPE)	걷힌 바이트를 가리키지 않게
안쪽 영역이 열린 동안 바깥 이름으로 깎기	거절(E-ALLOC-NESTED)	뿌리마다 커서는 하나
--target cortex_m + heap	거절(E-HEAP-NOHOST)	운영체제 없는 기계에는 힙이 없다
영역 밖 액터에게 영역 바이트를 send	E-ALLOC-OUTLIVES	액터가 바이트를 쥐고 영역보다 오래 살 수 있다 — 액터도 영역 안에서 만든다

표 18.3 — 영역의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

값은 지역·정적·얻은 것 가운데 한 곳에 살고, 얻는 뿌리는 자라지 않는 고정 창(alloc)과 자라는 힙(heap)이다. region <이름> <종류> do … end 가 영역을 열고, 블록을 나가는 모든 길에서 한꺼번에 되감긴다. 영역은 매개변수로 건넬 수 있다. 영역에서 얻은 바이트는 밖으로 들고 나갈 수 없고, 같은 뿌리의 안쪽 영역이 열린 동안 바깥에서 깎을 수 없다. 운영체제가 없는 기계에서는 힙이 거절된다.

19 소유 — 없앨 책임은 하나에게

이 장의 차례

19.1 owned 와 drop 221
19.2 갈래가 만나는 자리 223
19.3 해제와 완결 223
19.4 운영체제가 치워 주기를 기대하지 않는다 225
19.5 메모리 안전은 어느 강도로 보장되나 226
19.6 흔한 실수 226
19.7 이 장의 문법 한눈에 229

먼저 알아야 할 것

12장 빌리기 · 읽기 여럿 또는 쓰기 하나
18장 영역 · 영역은 끝날 때 한꺼번에 걷힌다
17장 실패를 설계하기 · 실패는 누군가 받아야 한다

돌아보기

18장의 영역은 값들을 한꺼번에 걷는다. 그런데 파일 핸들처럼 **하나씩**, 그리고 **닫다가 실패할 수 있는** 자원은 영역만으로 다룰 수 있는가?

> **답.** 영역은 바이트를 한꺼번에 되감을 뿐이라서, 파일을 닫거나 버퍼를 비우는 일은 하지 못한다. 그리고 닫기가 실패하면 그 실패를 받을 자리가 영역의 end 에는 없다. 그런 자원에는 값마다 없앨 책임을 지는 **소유**가 필요하다. 이 장이 그것을 다룬다.

이 장의 필요성과 맥락

메모리 결함의 오래된 두 이름은 **두 번 해제**와 **해제 누락**이고, 요즘 이름은 **옮긴 뒤 사용**이다. 셋은 모두 "이 값을 없앨 책임이 지금 누구에게 있는가" 에 답이 하나가 아닐 때 생긴다. Lowent 는 그 답이 언제나 하나이도록 번역에서 검사한다. 여기에 한 가지를 더한다 — 없애는 일이 **실패할 수 있으면** 조용히 없애지 않는다. 영역(18장)이 함께 죽는 값을 다뤘다면, 이 장은 따로 죽는 값을 다룬다.

이 장이 끝나면

owned t 가 무엇이고 drop 이 무엇을 하는지 알게 된다. 옮긴 값을 다시 쓰거나 두 번 없애면 거절되고, 갈래마다 소유 상태가 다르면 거절되는 이유를 익힌다. 없애기가 **해제**(실패 없음)와 **완결**(실패할 수 있음) 두 갈래라는 것, 완결이 필요한 값을 자동으로 버리면 왜 거절되는지 보게 된다. 이 언어의 메모리 안전이 어느 강도로 보장되는지도 정리한다.

이 장에서 답할 질문

1 소유가 없는 값은 넘길 때 어떻게 되는가?

19.1 owned 와 drop

`owned t` 는 소유를 가진 값이다. 소유를 가진 값은 **정확히 한 번** 없애져야 한다.

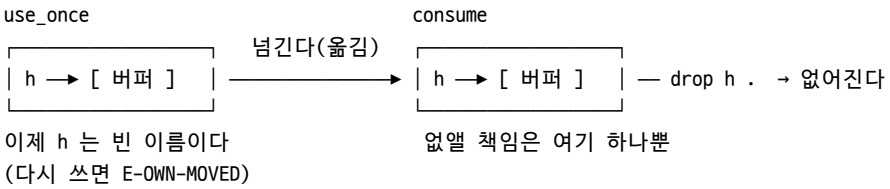
examples/ch19/sink.low

```
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
```

실행 결과

```
$ lowentc --run use_once sink.low 5
use_once(5) = 0
```

`consume` 은 `owned buffer` 를 받아 `drop h .` 로 없앤다. `use_once` 는 `var h owned buffer be v .` 로 소유 값을 만들어 `consume` 에 넘긴다. 넘기는 순간 소유가 **옮겨진다**. 이제 `h` 를 없앨 책임은 `consume` 에 있고, `use_once` 는 `h` 를 더 쓸 수 없다.



열쇠 하나짜리 사물함에 빗대면 쉽다. 열쇠를 건네면 내 손에는 열쇠가 없다. 사물함을 비울(없앨) 사람은 언제나 열쇠를 쥔 한 사람뿐이다. 그래서 «두 사람이 비운다»(두 번 해제)도, «아무도 안 비운다»(해제 누락)도, «건넨 열쇠로 또 연다» (옮긴 뒤 사용)도 생길 수 없다.

옮긴 값을 다시 쓰면 거절된다.

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
```

실행 결과

```
$ 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
```

두 번 없애도 거절된다.

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
```

실행 결과

```
$ 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
```

진단은 이것이 명세가 늘 오류라고 불러 온 자리이고 한때는 아무것도 강제하지 않았다고 적는다. 지금은 번역이 막는다.

> **문. 소유가 없는 값은 넘길 때 어떻게 되는가?**
>
> **답.** 베껴진다. `u64` 나 `point` 같은 값은 넘기면 복사되고 원래 이름도 계속 쓸 수 있다. 소유가 있는 값만 옮겨진다. 베끼기의 비용이 걱정되는 큰 값은 `ref` 로 빌려 넘긴다(12장). 그리고 값을 만드는 식이 놓일 자리가 비어 있고, 도중에 실패하지 않으며, 그 자리가 그 값만의 것이면 값은 목적지에 곧바로 만들어진다. 중간 임시값도, 옮기는 복사도 없다. 이 세 조건은 소스만 보고 셀 수 있다.

19.2 갈래가 만나는 자리

`if` 로 갈라졌다가 다시 만나는 자리에서, 소유 상태는 모든 길에서 같아야 한다.

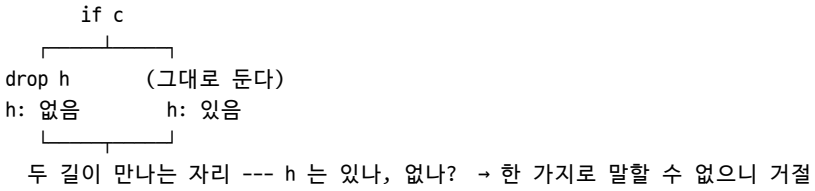
examples/ch19/join.low

```
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
```

실행 결과

```
$ 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`)
```

`c` 가 참인 길에서는 `h` 가 없어졌고 거짓인 길에서는 살아 있다. 두 길이 만난 뒤 `h` 가 살아 있는지를 아무도 말할 수 없다.



어떤 언어는 이 자리에 숨은 "없앴는지 표시하는 깃발" 을 두지만, Lowent 는 소유 상태가 **정적으로** 한 가지이기를 요구한다. 두 길 모두에서 없애거나, 두 길 모두에서 넘기면 된다.

같은 원리로, 묶음의 `owned` 칸 하나를 옮긴 뒤 묶음 전체를 다시 옮기면 거절된다(`E-OWN-PARTIAL`). 받는 쪽은 온전한 묶음을 받았다고 여기는데 그 안의 한 칸은 이미 남의 것이다.

19.3 해제와 완결

없애는 일은 두 갈래다.

갈래	예	다루는 법
해제	메모리를 돌려준다	실패가 없으므로 수명이 끝나는 자리에서 조용히 일어나도 된다
완결	파일을 닫는다, 버퍼를 비운다, 거래를 커밋한다	실패할 수 있으므로 저자가 적어서 불러야 한다

표 19.1 — 없애는 일의 두 갈래

해제는 삼킬 실패가 없다. 완결은 조용히 일어나면 그 실패를 건네줄 자리가 없다. 기준은 하나다 — **끝내는 일이 실패할 수 있는가.**

값의 종류	넘기면	끝나면
소유가 없는 값(u64 · point …)	베껴진다. 원래 이름도 계속 쓴다	할 일이 없다
owned, 해제만 필요	옮겨진다. 원래 이름은 빈다	수명이 끝나는 자리에서 조용히 해제된다
owned, 완결이 필요	옮겨진다. 원래 이름은 빈다	저자가 완결 op 을 적어서 불러야 한다(아니면 E-OWN-INCOMPLETE)

표 19.2 — 값이 넘겨질 때와 끝날 때 — 한눈에

어떤 타입이 완결을 요구하는지는 프로그램 자신이 선언한다. 그 타입을 owned 로 받아 result 를 돌려주는 op 이 있으면, 그것이 "이것을 끝내는 일은 실패할 수 있다" 는 선언이다. 새 낱말은 없다.

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 빈 일지는 쓸 자리가 없었다는 뜻이다 --- 그래서 이 갈래가 실제로 난다
  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
```

실행 결과

```
$ lowentc --run session complete.low 3
session(3) = ok 0
```

finish 가 owned journal 을 받아 result 를 돌려주므로 journal 은 완결이 필요한 타입이 된다. session 은 finish j 를 불러 끝내고, 그 결과(실패할 수도 있는)를 자기 결과로 돌려준다. 완결을 부르지 않고 범위를 벗어나게 두면 거절된다.

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
```

실행 결과

```
$ 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
```

진단의 말대로, 자동으로 없애는 것은 해제이고 해제에는 실패를 돌려줄 곳이 없으므로 그 실패를 **삼키게** 된다. 플러시·커밋·닫기 같은 실패할 수 있는 끝내기는 저자가 적어서 불러야 한다.

> **흔한 오해. 소멸자(destructor)가 알아서 파일을 닫아 주면 편하다**
>
> C++ 의 소멸자나 Rust 의 `Drop` 은 범위를 벗어날 때 자원을 닫는다. 닫기가 실패하면? 소멸자는 값을 돌려줄 수 없으므로 실패를 무시하거나 프로그램을 멈춘다. 디스크가 가득 차서 마지막 버퍼를 쓰지 못한 사실이 조용히 사라지는 자리다. Lowent 는 이 편의를 해제에만 허락하고, 완결은 적게 한다. 정말로 값과 그 실패를 함께 버리려면 `drop` 으로 **버린다고 말한다** — 진단도 그렇게 안내한다. 조용히 사라지는 것과 저자가 버리기로 적은 것은 다른 일이다.

19.4 운영체제가 치워 주기를 기대하지 않는다

많은 프로그램이 "어차피 프로세스가 끝나면 운영체제가 치운다" 에 기댄다. Lowent 는 그 기대를 하지 않는다. 없애기를 잊은 것을 번역에서 잡으므로, 운영체제가 없는 환경 — 프로세스가 끝나지 않는 펌웨어 — 에서도 같은 코드가 성립한다.

19.5 메모리 안전은 어느 강도로 보장되나

"메모리 안전" 이라는 말은 무엇이 어떻게 안전한지 밝혀야 뜻이 있다. 이 언어는 강도를 갈라 적는다.

규칙	강도	뜻
차용의 배타 · 참조 탈출 금지	정적 + 증명됨	번역이 막고, 규칙의 건전성은 Coq 로 증명되었다(순차 모델)
소유의 한 번 없애기 · 영역 탈출 금지	정적	번역이 막는다
슬라이스 경계 · 계약	동적	실행 중 검사하고, 증명되면 검사가 사라진다
세대 핸들의 늘어진 참조	동적	실행 중 세대를 비교한다(34장)

표 19.3 — 메모리 규칙과 그 강도

그러니 이 언어를 "전면 정적 안전" 이라고 부르면 틀린다. 정적으로 막는 것, 실행 중에 막는 것, 증명된 것을 섞은 설계다. 무엇이 증명되었고 증명과 컴파일러 사이에 어떤 틈이 있는지는 42·50장이 다룬다.

19.6 흔한 실수

반례. 반복 안에서 소유 값을 넘긴다

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 ✘ 첫 바퀴에서 옮겨 간 `h` 를 둘째 바퀴가 또 넘긴다
    let a u8 be consume h .
    set i (add i 1) .
  end
  return 0 .
end
```

실행 결과

```
$ 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
```

첫 바퀴에서 `consume h` 가 소유를 가져가면 둘째 바퀴의 `h` 는 이미 남의 것이다. 번역은 바퀴를 하나씩 따라가지 않고도 "바퀴 몸이 시작할 때와 다른 모양으로 끝난다" 를 보고 `E-OWN-MOVED` 로 거절한다. 고치는 길은 둘이다. 넘기기를 반복 밖으로 빼거나, 넘긴 자리를 `set` 으로 새 값으로 다시 채워 바퀴가 같은 모양으로 끝나게 한다.

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 넘긴 자리를 새 값으로 다시 채운다 --- 바퀴가 시작할 때와 같은 모양으로 끝난다
    set h v .
    set i (add i 1) .
  end
  drop h .
  return n .
end
```

실행 결과

```
$ lowentc --run three_times loopmove_fixed.low 5
three_times(5) = 3
```

반례. 읽기만 했다고 생각하고 옮긴다

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 ✘ 값을 꺼내 담는 것도 옮기기다 --- 그 뒤의 `drop h` 는 옮긴 값을 없앤다
  let v u8 be h .
  drop h .
  return v .
end
```

실행 결과

```
$ 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 .` 는 `h` 를 들여다보는 것이 아니라 `v` 로 **옮기는** 것이다. 소유가 있는 값은 이름에 담는 순간 옮겨 가므로 뒤의 `drop h` 는 옮긴 값을 없애려는 셈이다. 들여다보기만 하려면 같은 op 안에서 `ref h` 로 빌린다. 진단이 적은 대로 op 경계를 넘는 빌림은 이 판에서 아직 낮춰지지 않는다.

반례. 다른 이름에 담으면 사본이 생긴다고 믿는다

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 ✘ 다른 이름에 담으면 사본이 생긴다고 믿었다 --- 소유는 `h2` 로 옮겨 갔다
  var h2 owned buffer be h .
  let a u8 be consume h2 .
  let b u8 be consume h .
  return 0 .
end
```

실행 결과

```
$ 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
```

`u64` 같은 값은 다른 이름에 담으면 베껴진다. 소유가 있는 값은 베껴지지 않고 옮겨 간다. 사본이 둘이면 누가 없앨지 정할 수 없고, 둘 다 없애면 두 번 없애기가 되기 때문이다. `var h2 owned buffer be h .` 뒤로 소유자는 `h2` 하나다.

> **흔한 오해. `drop` 을 적지 않으면 샌다**

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 `c` 가 거짓인 길은 `drop` 없이 떠난다 --- 해제는 실패가 없으므로 수명이 끝나는 자리
에서 조용히 일어난다
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
```

실행 결과

```
$ 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
```

`c` 가 0 인 길은 `drop` 없이 떠나지만 거절되지 않고 새지도 않는다. 메모리를 돌려주는 **해제**는 실패할 수 없으므로 수명이 끝나는 자리에서 조용히 일어난다. 저자가 반드시 적어야 하는 것은 실패할 수 있는 **완결**뿐이다(`incomplete.low`). 다만 갈래가 **다시 만나는** 자리에서는 소유 상태가 같아야 한다(`join.low`). 떠나 버리는 길은 만나지 않으므로 해당하지 않는다.

19.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`input h owned buffer .` · `var h owned buffer be v .`	소유를 가진 값	없앨 책임이 이름 하나에 있다
`consume h`	넘기면 소유가 옮겨 간다	옮긴 뒤에는 쓸 수 없다 — `E-OWN-MOVED`
`drop h .`	지금 없앤다고 적는다	두 번 없애기는 거절된다
`set h v .` **(옮긴 뒤)**	옮겨 간 자리를 다시 채운다	반복 몸이 같은 모양으로 끝난다
`if` **뒤 갈래마다 다른 소유 상태**	거절(`E-OWN-JOIN`)	숨은 “없앴나” 깃발을 두지 않는다
`fn finish input j owned journal . output result …`	이 타입의 끝내기는 실패할 수 있다는 선언	새 낱말 없이 완결을 선언한다
완결을 부르지 않고 범위를 떠남	거절(`E-OWN-INCOMPLETE`)	끝내기의 실패를 삼키지 않는다
완결이 필요 없는 값을 적지 않고 떠남	조용히 해제된다	해제는 실패가 없다

표 19.4 — 소유의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`owned t` 는 정확히 한 번 없애져야 하고, 넘기면 옮겨진다. 옮긴 값을 다시 쓰거나 두 번 없애면, 그리고 갈래마다 소유 상태가 다르면 거절된다. 없애기는 실패 없는 해제와 실패할 수 있는 완결로 나뉘며, 완결이 필요한 타입은 그것을 `owned` 로 받아 `result` 를 돌려주는 op 이 선언한다. 완결이 필요한 값을 자동으로 버리면 거절된다. 메모리 규칙은 정적·동적· 증명됨으로 강도를 갈라 이해한다.

20 할당기와 고정 메모리

이 장의 차례

20.1 할당기를 이루는 셋 232
20.2 빌린 바이트를 잘라 준다 232
20.3 할당기를 갈아 끼운다 233
20.4 뿌리에서 곧장 깎는 기본 할당기 236
20.5 조각을 늘리고 돌려준다 — `grow` 와 `release` 237
20.6 표준 라이브러리에서 셋은 어디에 있나 239
20.7 고정 창의 크기는 누가 정하나 239
20.8 비트는 그대로, 읽는 법만 240
20.9 흔한 실수 241
20.10 이 장의 문법 한눈에 245

먼저 알아야 할 것

18장 영역 · 고정 창과 힙, 두 뿌리
16장 권한 · 권한은 종류로 인가되고 실행 비용이 없다
15장 효과 · `via` 는 타입 인자가 할당 효과를 정하게 한다

돌아보기

18장에서 운영체제가 없는 기계를 대상으로 지으면 무엇이 거절되었고, 무엇은 여전히 쓸 수 있었는가?

> **답.** `heap` 효과·`cap heap` 입력·`region … heap` 블록이 모두 `E-HEAP-NOHOST` 로 거절되었다. 자라지 않는 고정 창에서 깎는 `alloc` 은 여전히 쓸 수 있었다. 이 장은 그 고정 창 위에서, 그리고 남이 빌려준 바이트 위에서 자리를 나눠 주는 **할당기**를 다룬다.

이 장의 필요성과 맥락

영역은 "블록이 끝날 때 한꺼번에" 라는 한 가지 정책이다. 실제 프로그램은 더 많은 정책을 원한다. 커다란 버퍼 하나를 받아 앞에서부터 잘라 쓰고 싶고, 정렬을 맞춰 자르고 싶고, 같은 코드를 힙에서도 고정 창에서도 돌리고 싶다. Lowent 에서 할당기는 언어의 특별한 장치가 아니라 **트레이트를 갖춘 액터**다. 그래서 사용자가 만든 할당기와 표준 할당기가 같은 자리에 들어간다. 메모리를 다루는 제5부의 마지막 장으로, 운영체제 없는 기계까지 이어지는 고정 메모리의 길을 정리한다.

이 장이 끝나면

할당기가 권한·정책·상태 셋으로 이루어진다는 것을 알게 된다. 빌린 바이트를 잘라 주는 범프 할당기를 쓰고, 메모리 부족이 값이라는 것을 확인한다. 할당기를 타입 매개변수로 받고 `using` 으로 건네는 법, 뿌리에서 곧장 깎는 기본 할당기와 권한 칸의 규칙을 익힌다. 운영체제 없는 기계의 고정 창 크기를 링커가 정하는 방식과, 비트를 그대로 둔 채 읽는 법만 바꾸는 `bit_cast` 도 보게 된다.

이 장에서 답할 질문

1 범프 할당기로 조각 하나를 돌려줄 수 있는가?

20.1 할당기를 이루는 셋

층	언제 있나	무엇인가
권한	번역할 때만	뿌리에 닿아도 되는가 — `cap allocator` · `cap heap`
정책	번역할 때(타입)	어떤 규칙으로 깎는가 — `byte_allocator` 트레이트를 갖춘 **타입**
상태	실행 중	커서와 받침 바이트 — 그 타입의 **값**(액터)

표 20.1 — 할당기를 이루는 셋

표준 라이브러리 `allocs` 의 트레이트 `byte_allocator` 는 세 op 을 요구한다. `reserve n` 은 `n` 바이트를 잘라 `option` 으로 주고, `grow` 는 마지막 조각을 제자리에서 늘리며, `used` 는 지금까지 쓴 양을 답한다. 액터(25장)는 자기 상태를 들고 `send` 로만 말을 거는 객체다. 여기서는 `spawn actor` 로 만들고 `send a reserve 3` 으로 부른다는 것만 알면 된다.

20.2 빌린 바이트를 잘라 준다

가장 단순한 할당기는 **범프**다. 커서 하나를 두고, 달라는 만큼 잘라 준 뒤 커서를 민다.

```
buf (8 바이트)
[ a  a  a | b  b | ·  ·  · ]
  └ reserve 3 이 준 조각
          └ reserve 2 가 준 조각
                ▲ 커서 (used = 5) --- 다음 reserve 는 여기서부터 자른다
```

`reserve 4` 를 청하면 남은 것은 3 바이트뿐 → `none` (모자람도 값이다)

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
```

실행 결과

```
$ 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` 로 할당기를 만들고, `send a init buf` 로 잘라 줄 원본 바이트를 건다. 이 할당기는 스스로 메모리를 만들지 못한다. 몰래 할당하지 않는 규율이다.

- `send a reserve 3` 은 3 바이트를 잘라 준다. 돌려받은 것은 복사본이 아니라 원본의 일부를 가리키는 슬라이스라서, `set (index pv 0) 65 .` 가 호출자의 `buf` 첫 바이트를 바꾼다. VM 이 보여 주는 인자 `[65,0,…]` 가 그 흔적이다.
- `send a reserve 99` 는 자리가 모자라 `none` 을 준다. 트랩이 아니다. **메모리 부족은 값**이고, 부르는 쪽이 검사한다.

이 op 의 머리에는 `cap allocator` 도 `alloc` 도 없다. 효과는 `state` 뿐이다. 바이트는 호출자가 빌려주었고, 할당기는 그것을 나눠 줄 뿐이기 때문이다. 권한이 능력을 가른다 — 할당 권한을 받지 않은 코드도 남이 준 바이트 위에서는 온전히 할당한다.

> **문. 범프 할당기로 조각 하나를 돌려줄 수 있는가?**
>
> **답.** 마지막 조각 하나만 돌려받는다(`release`, 트레이트 `freeing_allocator`). 그 밖의 조각은 `false` 를 답하고 아무것도 바꾸지 않는다. 모르는 조각을 받아 주는 척하면 두 번 돌려주기가 남의 자리를 지우기 때문이다. 받고 놓기를 되풀이하는 모양이면 세대 핸들을 쓰는 `pool` 이 맞다(35장).

20.3 할당기를 갈아 끼운다

할당기를 쓰는 코드는 어느 구현인지 몰라도 된다. 할당기의 **타입**을 번역 시점 매개변수로 받고, 그 **값**은 `using` 절로 받는다.

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
```

실행 결과

```
$ 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]
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]) = 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]
```

- `input comptime a type .` 은 할당기의 타입이다. 번역할 때 구체 타입으로 정해진다(22장).
- `using al a .` 은 그 타입의 할당기 값을 `al` 이라는 이름으로 받는다. `using` 은 입력이 아니다. 부르는 쪽은 인자 자리에 적지 않고, 바인딩에 `let n u64 using b be two_from .` 으로 적는다.
- `requires allocs.byte_allocator a .` 는 `a` 가 트레이트를 갖추어야 한다는 조건이다(23장).
- `effects state via a .` 는 할당기의 `reserve` 가 내는 효과가 곧 이 op 의 효과라는 뜻이다.

같은 `two_from` 에 `bump_bytes` 를 주면 3 + 5 = 8 을 쓰고, 8 의 배수로 시작점을 맞추는 `bump_aligned` 를 주면 둘째 조각이 8 에서 시작해 13 을 쓴다. 번역할 때 타입이 확정되므로 가상 함수 표도 간접 호출도 없다. 갈아 끼우기의 실행 비용이 0 이다.

출처를 적지 않으면 기본값이 정해진다. 바인딩의 `using`, 그 op 의 `using` 이름, 그리고 타입이 맞는 입력·바인딩이 **하나뿐**이면 그것을 쓴다. 둘 이상이면 짐작하지 않고 `E-ALLOC-AMBIGUOUS` 로 적으라고 한다. 기본값은 op 경계를 넘지 않는다. 부른 쪽의 할당기가 저절로 흘러드는 일이 없으니, **전역 할당기는 없다.**

차례	출처	왜 이 자리인가
1	바인딩에 적은 `using <이름>`	적은 것이 언제나 이긴다
2	이 op 의 `using` 절 이름 가운데 타입이 맞는 **유일한** 것	op 이 스스로 받겠다고 한 할당기
3	이 op 의 입력·바인딩 가운데 타입이 맞는 **유일한** 것	눈에 보이는 값이 하나뿐이면 헷갈릴 일이 없다
없음	`E-ALLOC-NOSOURCE`	전역 할당기로 메우지 않는다
둘 이상	`E-ALLOC-AMBIGUOUS`	짐작하지 않고 후보 이름을 모두 말한다

표 20.2 — 할당기 출처를 정하는 차례 — 위에서 처음 맞는 것

필드는 세지 않는다. 구조체 안에 숨은 할당기가 조용히 쓰이면, 어느 버퍼가 줄어드는지 코드를 읽어서 알 수 없기 때문이다.

출처가 하나도 없으면 이렇게 거절된다.

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 ✘ 이 op 안에는 깎아 올 할당기가 하나도 없다
  let n u64 be one_from .
  return n .
end
```

실행 결과

```
$ 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 에는 입력도, 띄운 액터도, using 절도 없다. 다른 언어라면 여기서 전역 힙이 조용히 쓰였을 것이다. Lowent 는 할당기를 입력으로 받거나, 이 자리에서 만들거나, using 절을 적으라고 한다. 호출이 거절되었으니 caller 의 state 도 실제로 쓰이지 않아 W-EFFECT-OVER 가 함께 붙는다. 첫 오류를 고치면 사라진다.

거꾸로, 할당기를 쓰지 않는 호출에 using 을 적어도 거절된다.

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 는 할당기를 쓰지 않는데 using 으로 골라 준다
  let n u64 using b be twice 4 .
  return n .
end
```

실행 결과

```
$ 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 는 수를 두 배로 할 뿐 할당기를 받지 않는다. 쓰이지 않을 선택을 적어 두면 읽는 사람은 twice 가 메모리를 쓴다고 오해한다. 그래서 적은 것은 반드시 쓰여야 한다.

20.4 뿌리에서 곧장 깎는 기본 할당기

allocs 는 뿌리에서 곧장 깎는 할당기 둘도 같은 트레이트로 낸다.

액터	상태	reserve 의 효과
fixed_bytes	root cap allocator . · 쓴 양	alloc state — 어디서나
heap_bytes	root cap heap . · 쓴 양	heap state — 운영체제가 있는 기계만

표 20.3 — 기본 할당기

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
```

실행 결과

```
$ lowentc --run main fixed.low
main() = 128
```

fixed_bytes 의 상태에는 **권한 칸**이 있다. 권한은 실행 중 값이 아니므로 그 칸은 크기가 없다. 대신 규칙이 하나 붙는다. 권한 칸을 가진 액터는 **같은 종류의 권한을 쥔 op** 에서만 띄울 수 있다.

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
```

실행 결과

```
$ 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 는 cap heap 을 받지 않았는데 heap_bytes 를 띄우려 했다. 이것이 허락되면 권한 없는 곳에서 한 줄로 힙을 지어낼 수 있다. 권한은 건네받는 것이지 주워 쓰는 것이 아니다.

권한을 받으면 같은 일이 된다.

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 고정 창(기본 64 KiB)보다 큰 100000 바이트도 힙은 준다
  let a option mut slice u8 be send hb reserve 100000 .
  guard is_some a . else return 1 .
  rem 쓴 양이 100000 이 아니면 2, 맞으면 0 을 돌려준다
  guard eq (send hb used) 100000 . else return 2 .
  return 0 .
end
```

실행 결과

```
$ lowentc --run main heapbytes.low
main() = 0
```

main 이 cap heap 을 받았고 효과 줄에 heap 을 적었다. 그래서 heap_bytes 를 띄울 수 있고, 고정 창보다 큰 100000 바이트도 받는다. 힙은 모자라면 청크를 더 잇는다. 운영체제가 없는 기계를 대상으로 지으면 이 파일은 E-HEAP-NOHOST 로 거절된다(18장).

20.5 조각을 늘리고 돌려준다 — grow 와 release

범프 할당기는 앞으로만 민다. 그래도 **마지막에 준 조각**만은 되돌려도 안전하다. 그 뒤로 아무도 자리를 받지 않았기 때문이다. grow 는 그 조각을 제자리에서 늘리고, release(트레이트 freeing_allocator)는 그 조각을 돌려받는다.

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 방금 받은 조각을 4 에서 6 바이트로 제자리에서 늘린다
  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 는 마지막 조각이 아니다(뒤에 qv 가 있다) — 돌려받지 않는다
  let back1 bool be send b release gv .
  rem qv 는 마지막 조각이다 — 돌려받는다
  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 쓴 양 · 첫 답 · 둘째 답을 세 자리 수로 묶어 돌려준다
  return add (mul u 100) (add (mul d1 10) d2) .
end
```

실행 결과

```
$ 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` 은 4 바이트였던 `pv` 를 6 바이트로 늘린다. 늘리는 대상은 크기가 아니라 **조각 자체**다. 구현은 내장 `same_slice a b`(시작 주소와 길이가 같은가)로 `pv` 가 방금 준 바로 그 바이트인지 확인한다. 길이만 같은 남의 버퍼를 넘기면 `none` 이다. 크기만으로 알아보면 두 그릇이 조용히 겹치기 때문이다.
- `qv` 를 받은 뒤에는 `gv` 가 더 이상 마지막이 아니다. 그래서 `release gv` 는 `false` 이고 아무것도 바꾸지 않는다.
- `release qv` 는 `true` 다. 커서가 6 으로 돌아가므로 `used` 가 6 이다. 답 601 은 “쓴 양 6 · 첫 답 거짓 · 둘째 답 참” 이다.

두 op 모두 **최적화이지 약속이 아니다**. 못 늘리면 부르는 쪽이 새로 받아 복사하면 되고, 답은 같아야 한다. 뿌리에서 곧장 깎는 `fixed_bytes·heap_bytes` 의 `grow` 는 언제나 `none` 이다. 뿌리는 마지막 조각이 누구 것인지 모른다.

20.6 표준 라이브러리에서 셋은 어디에 있나

모듈 · 이름	권한	정책(타입)	상태(값)
`allocs.bump_bytes` · `bump_aligned`	없다 — 빌린 바이트	`byte_allocator` · `freeing_allocator` 를 갖춘 액터	받침 바이트 · 커서
`allocs.fixed_bytes` · `heap_bytes`	권한 칸 `cap allocator` · `cap heap`	`byte_allocator`	쓴 양
`vecgen.vec t a`	없다	할당기 타입 `a` 를 매개변수로 받는다	원소 수 · 버퍼 — 할당기는 `open` 이 `using` 으로 받는다
`growvec.gvec`	없다	`vecgen.vec u8 allocs.bump_bytes` 로 고정한 이름	`vecgen` 과 같다
`pool.block_pool b`	없다 — 빌린 바이트	할당기가 아니다 — 세대 핸들로 블록을 낱낱이 돌려받는 구조체	블록 · 세대 배열

표 20.4 — 할당에 쓰는 모듈 — 권한 · 정책 · 상태 가운데 무엇을 맡나

권한은 뿌리에 닿는 두 액터에만 있다. 나머지는 모두 누군가 건네준 바이트 위에서 일한다. 그래서 권한을 받지 않은 라이브러리 코드도 그릇을 만들고 키울 수 있고, 어느 코드가 새 메모리를 프로그램에 들여오는지는 권한을 받은 op 만 보면 안다.

20.7 고정 창의 크기는 누가 정하나

운영체제가 없는 기계에서 고정 창은 링커가 정한 두 기호 사이의 메모리다. 크기는 실행 파일에 박히지 않는다. 컴파일러가 링커 스크립트 조각을 내주고, 보드마다 그 조각의 크기만 바꾼다.

```
$ lowentc --emit-ldscript --fixed-bytes 4096 fixed.low
.lw_fixed (NOLOAD) : ALIGN(8)
{
    __lw_fixed_start = .;
    . = . + 4096;
    __lw_fixed_end = .;
}
```

호스트에서 다른 보드를 흉내 낼 때는 VM 에 창의 크기를 준다. `fixed.low` 는 64 바이트를 두 번 청하므로, `lowentc --fixed-bytes 100 --run main fixed.low` 로 창을 100 바이트로 줄이면 둘째 `reserve` 가 `none` 이 되어 `main() = 2` 가 나온다. 작은 기계에서 어디서 메모리가 모자라는지를 개발 기계에서 먼저 볼 수 있다.

> **흔한 오해. 임베디드에서는 동적 할당을 쓰지 않으니 할당기가 필요 없다**
>
> "동적 할당 금지" 의 속뜻은 **자라는 힙**과 **실패를 숨기는 할당**을 쓰지 말라는 것이다. 부팅 때 한 번 정해진 창에서 깎고, 부족을 값으로 돌려주고, 한꺼번에 되감는 할당은 그 규율과 충돌하지 않는다. Lowent 의 고정 창은 크기가 링커에 적히고, 부족은 `none` 이며, 힙 효과는 번역에서 막힌다. 규칙을 문법이 지키는 것이다.

20.8 비트는 그대로, 읽는 법만

같은 바이트를 다른 타입으로 읽는 일은 두 갈래다. 바이트 슬라이스 위에 구조체 배치를 얹는 `view` 는 13장에서 보았다. 폭이 같은 두 스칼라 사이에서 비트열을 그대로 두고 읽는 법만 바꾸는 것은 `bit_cast` 다.

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
```

실행 결과

```
$ lowentc --run reinterpret bitcast.low -1
reinterpret(-1) = 4294967295
$ lowentc --run float_bits bitcast.low 1.0
float_bits(1.0) = 4607182418800017408
```

`i32` 의 −1 은 모든 비트가 1 이므로 `u32` 로 읽으면 4294967295 다. `f64` 의 1.0 을 `u64` 로 읽으면 IEEE 754 표현이 그대로 나온다. `cast` 와 달리 값을 옮기지 않으므로 멈추는 일도 없다.

목표 타입은 **모든 비트열이 쓸모 있는 값**이어야 한다. `bool` 과 `enum` 은 그렇지 않다.

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
```

실행 결과

```
$ 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
```

`u8` 의 2 를 `bool` 로 읽으면 참도 거짓도 아닌 값이 된다. 그런 값이 만들어지는 길을 막는다.

20.9 흔한 실수

반례. 범프 할당기에 `init` 을 잊는다

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` 를 빠뜨렸다 --- 잘라 줄 바이트가 없는 할당기에 청한다
  let p option mut slice u8 be send a reserve 3 .
  guard is_some p . else return 91 .
  return send a used .
end
```

실행 결과

```
$ 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` 는 스스로 메모리를 만들지 않는다. 잘라 줄 바이트를 `send a init buf` 로 걸기 전에는 나눠 줄 것이 없다. 갓 띄운 액터의 상태 칸은 모두 0 이고, 0 은 슬라이스가 아니다 — 그 자리를 읽으면 VM 과 네이티브가 서로 다르게 돈다. 그래서 번역이 거절한다 (`E-ACTOR-UNINIT`). 할당기를 띄우는 줄 바로 다음에 `init` 을 둔다.

반례. 같은 바이트를 두 할당기에 건다

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 ✘ 같은 바이트를 두 할당기에 걸었다 --- 두 할당기가 같은 자리를 나눠 준다
  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 를 썼는데 66 이 나온다
  return index pv 0 .
end
```

실행 결과

```
$ 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
```

두 할당기는 서로를 모른다. 둘 다 `buf` 의 앞에서부터 잘라 주므로 `pv` 와 `qv` 가 같은 자리가 되고, 65 를 쓴 뒤 66 을 쓰면 `pv` 를 읽어도 66 이다. 쓰기 빌림은 하나여야 한다는 규칙(12장)이 액터 경계에서도 서야 하므로 `E-EXCL` 로 거절한다 (2026-09-16 까지는 통과했다). 할당기마다 따로 된 바이트를 건다. 한 버퍼를 나눠야 하면 `subslice` 로 겹치지 않는 두 조각을 만든다.

반례. 맞는 할당기가 둘인데 `using` 을 적지 않는다

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 ✘ 맞는 타입의 할당기가 둘인데 어느 것인지 적지 않았다
  let n u64 be two_from .
  return n .
end
```

실행 결과

```
$ 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` 와 `g` 가 모두 `bump_bytes` 라서 도구가 짐작할 수 없다. 짐작하면 작은 버퍼에서 깎아야 할 것을 큰 버퍼에서 깎거나 그 반대가 되고, 그런 결함은 메모리가 넉넉한 개발 기계에서는 드러나지 않는다. 그래서 `E-ALLOC-AMBIGUOUS` 로 멈추고 적으라고 한다.

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 바인딩에 `using` 으로 출처를 적는다
  let n u64 using g be two_from .
  let m u64 using g be two_from .
  return m .
end
```

실행 결과

```
$ 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]
```

두 번 부른 `two_from` 이 같은 `g` 에서 3 바이트씩 깎았으므로 `used` 가 6 이다. 출처가 호출마다 적혀 있으니 어느 버퍼가 줄어드는지 읽어서 안다.

반례. 제네릭 op 에서 `via a` 를 빠뜨린다

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 ✘ `via a` 가 없다 --- 할당기가 내는 `alloc` 이 이 op 의 선언에 들어오지 않는다
  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
```

실행 결과

```
$ 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` 은 `effects state .` 만 적었지만, `fixed_bytes` 로 단형화하면 `reserve` 가 `alloc` 을 낸다. `via a` 가 있어야 "타입 `a` 가 내는 할당 계열 효과도 내 선언이다" 가 되어 인스턴스마다 효과가 정확해진다. 없으면 `E-EFFECT` 가 나고, 효과가 부르는 쪽에 번지지 않아 `with_fixed` 에는 엉뚱하게 `W-EFFECT-OVER` 까지 붙는다. 첫 오류를 고치면 둘째도 사라진다.

흔한 오해. 할당기를 하나 더 띄우면 창도 하나 더 생긴다

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 할당기 둘이지만 고정 창은 하나다
  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` 는 아직 아무것도 쓰지 않았지만 창에 남은 자리가 모자라다
  let b option mut slice u8 be send f2 reserve 40000 .
  guard is_some b . else return 2 .
  return 0 .
end
```

실행 결과

```
$ lowentc --run main shared_window.low
main() = 2
```

`fixed_bytes` 는 **뿌리**에서 깎는다. 뿌리는 하나이고 커서도 하나다(18장). `f1` 이 40000 바이트를 가져가면 `f2` 는 아직 아무것도 쓰지 않았어도 기본 창(65536 바이트)에 남은 자리가 모자라 `none` 을 받는다. 할당기 값의 `used` 는 **그 할당기가** 쓴 양이지 창 전체의 남은 양이 아니다. 따로 된 예산이 필요하면 창에서 한 번 크게 받아 `bump_bytes` 여럿에 겹치지 않게 나눠 건다.

20.10 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`var a allocs.bump_bytes be spawn actor allocs.bump_bytes .`	할당기(상태)를 띄운다	상태는 액터 값 — 전역 할당기가 없다
`send a init buf`	잘라 줄 바이트를 건다	할당기는 몰래 메모리를 만들지 않는다
`send a reserve 3 • send a used`	조각을 청한다(option) · 쓴 양	부족은 트랩이 아니라 값
`input comptime a type .`	할당기의 타입(정책)을 번역 때 받는다	갈아 끼우기의 실행 비용이 0
`using al a .`	그 타입의 할당기 값을 받는다 — 입력이 아니다	부르는 자리의 인자에 끼지 않는다
`let n u64 using g be two_from .`	이 호출이 깎을 할당기를 적는다	둘 이상이면 짐작하지 않는다
`effects state via a . • requires allocs.byte_allocator a .`	타입의 효과를 물려받는다 · 트레이트 조건	인스턴스마다 효과가 정확하다
`allocs.fixed_bytes` `• allocs.heap_bytes`	뿌리에서 곧장 깎는 기본 할당기	같은 종류의 권한을 쥔 op 만 띄운다 — `E-CAP-FORGE`
`send b grow pv 6 • send b release qv`	마지막 조각을 늘린다 · 돌려받는다	크기가 아니라 `same_slice` 로 정체를 확인한다
출처 없음 · 쓰이지 않는 `using`	`E-ALLOC-NOSOURCE` · `E-ALLOC-USING-UNUSED`	전역 할당기도, 헛된 선택도 없다
`bit_cast u32 x`	비트는 그대로 두고 읽는 법만 바꾼다	`bool`·`enum` 으로는 읽지 않는다

표 20.5 — 할당기의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

할당기는 권한·정책(트레이트를 갖춘 타입)·상태(액터 값) 셋이다. 범프 할당기는 빌린 바이트를 잘라 주고 부족을 `none` 으로 알린다. 할당기는 `input comptime a type .` 과 `using al a .` 로 받아 갈아 끼우며 실행 비용이 없고, 전역 할당기는 없다. `fixed_bytes·heap_bytes` 는 권한 칸을 가진 기본 할당기이고, 같은 종류의 권한을 쥔 op 만 띄울 수 있다. 고정 창의 크기는 링커가 정한다. `bit_cast` 는 비트를 그대로 두고, `bool·enum` 으로는 읽지 않는다.

제6부 — 추상

21 모듈 — 감춘 것이 기본이다

이 장의 차례

21.1 파일 하나, 모듈 하나 249
21.2 들여오기 250
21.3 무엇을 내보낼 수 있고, 어떻게 줄여 부르나 251
21.4 검색 경로가 없다 252
21.5 이름이 부딪히면 253
21.6 최상위 선언의 차례는 무관하다 253
21.7 흔한 실수 254
21.8 이 장의 문법 한눈에 259

먼저 알아야 할 것

3장 겉모습 · 점이 붙은 이름은 선언된 이름을 가리키는 경로다
16장 권한 · 공개된 op 머리만 보면 라이브러리가 닿는 바깥을 안다

돌아보기

16장의 마지막 사례에서, 들여온 라이브러리가 네트워크에 닿는지 알려면 무엇을 보면 된다고 했는가?

> **답.** 그 라이브러리가 **내보낸** op 의 머리를 훑어 `cap net`(그리고 `cap c·cap machine`)을 받는 op 이 있는지 보면 된다고 했다. 이 장은 그 "내보낸" 이 무엇이고, 다른 파일의 이름을 어떻게 들여오는지를 다룬다.

이 장의 필요성과 맥락

프로그램이 파일 하나를 넘어서면 두 가지 물음이 생긴다. **무엇이 밖에서 보이는가**, 그리고 **이 파일이 무엇에 기대는가**. 첫 물음의 답이 흐리면 내부 구현을 고칠 때마다 누가 깨질지 모르고, 둘째 물음의 답이 흐리면 같은 소스가 기계에 따라 다른 것을 가져온다. Lowent 는 둘 다 소스에 적게 한다. 제6부 "추상" 의 첫 장으로, 코드를 나누는 가장 바깥 단위부터 세운다.

이 장이 끝나면

파일 하나가 모듈 하나이고 `export` 를 붙인 것만 밖에서 보인다는 것을 알게 된다. `use … from "자리"` 로 다른 파일을 들여오고, 여러 파일을 한 번역 단위로 넘기며, 표준 라이브러리 이름을 들여오는 법을 익힌다. 모듈 이름으로 한정해도 감춘 이름에는 닿지 못한다는 것, 검색 경로가 없다는 설계, 최상위 선언의 차례가 무관하다는 것도 보게 된다.

이 장에서 답할 질문

1 `from "…"` 없이 `use geom .` 이라고만 적으면 어디서 찾는가?

21.1 파일 하나, 모듈 하나

소스 파일 하나가 모듈 하나다. 모듈은 첫 줄에 자기 이름을 적는다. 모듈 이름은 파일 이름과 같을 필요가 없다.

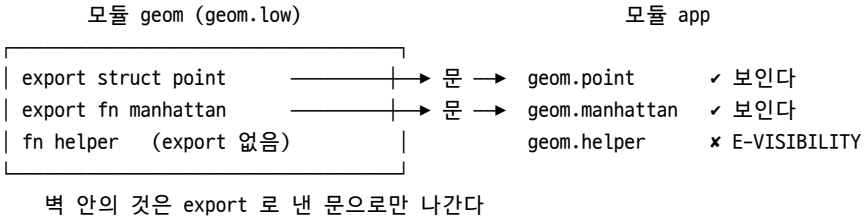
examples/ch21/geom.low

```
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
```

실행 결과

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

`geom` 모듈은 `point` 구조체와 `manhattan` op 을 `export` 로 내보낸다. `helper` 에는 `export` 가 없다. 그래서 `helper` 는 이 모듈 안에서만 쓸 수 있다.



감춘 것이 기본이다. 밖에서 보이는 것은 곧 약속이고, 실수로 약속하는 일보다 실수로 감추는 일이 고치기 쉽다. 네이티브로 낼 때도 `export` 한 op 만 C 에서 부를 수 있는 심볼이 되고 나머지는 모두 `static` 이다.

21.2 들여오기

`use <모듈> from "<자리>" .` 은 그 자리의 파일에서 모듈을 들여온다. 자리는 **선언한 파일을 기준으로 한 경로**다.

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
```

실행 결과

```
$ lowentc --run distance app.low 3 4
distance(3, 4) = 7
```

들여온 이름은 모듈 이름으로 한정해 부른다 — `geom.point`, `geom.manhattan`. 붙은 점은 값에 대한 연산이 아니라 선언된 이름을 가리키는 경로다(3장).

한정해도 감춘 이름에는 닿지 못한다.

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
```

실행 결과

```
$ 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)
```

진단의 말대로 "아무것도 감출 수 없는 모듈은 모듈이 아니라 접두사다".

> **문.** `from "…"` 없이 `use geom .` 이라고만 적으면 어디서 찾는가?
>
> **답.** 같은 번역 단위 안에서 찾는다. `lowentc --check app.low geom.low` 처럼 파일을 여럿 넘기면 그 파일들이 하나의 단위가 된다. 그 단위에 없으면 컴파일러는 `W-USE-EXTERNAL` 로 "이 모듈이 있는지 확인할 수 없다" 고 말하고, 그 이름을 실제로 쓰면 거절한다. 표준 라이브러리 이름(`use allocs .`)은 설치된 표준 모듈 자리에서 해소된다.

21.3 무엇을 내보낼 수 있고, 어떻게 줄여 부르나

`export` 는 이름을 가진 선언 대부분에 붙는다. 붙지 않는 것도 있다.

선언	export	까닭
`fn` · `proc`	된다	다른 모듈이 부를 op
`struct` · `enum` · `type` · `newtype`	된다	내보낸 op 의 입력·출력 타입도 함께 내보내야 쓸 수 있다
`trait` · `actor`	된다	다른 모듈의 타입이 갖출 약속 · 다른 모듈이 띄울 액터
최상위 `let`(상수)	안 된다 — `E-TOPLEVEL`	상수는 모듈 안에 둔다. 밖에 알려야 하면 그 값을 돌려주는 `fn` 을 내보낸다
`test`	안 된다 — `E-TOPLEVEL`	시험은 그 모듈을 짓는 사람의 것이다

표 21.1 — `export` 를 붙일 수 있는 선언

모듈 이름이 길거나 두 모듈의 이름이 같으면 `as` 로 줄여 부른다.

examples/ch21/alias.low

```
module alias .
rem run: distance 3 4

rem 들여온 모듈을 짧은 이름 g 로 부른다
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
```

실행 결과

```
$ lowentc --run distance alias.low 3 4
distance(3, 4) = 7
```

`use geom from "geom.low" as g .` 뒤로는 `g.point`·`g.manhattan` 으로 부른다. 별칭은 이 파일 안에서만 쓰는 이름이고, 모듈의 실제 이름(`geom`)은 바뀌지 않는다. 다만 **이 파일 안에서는** 원래 이름이 서지 않는다 — `geom.point` 라고 적으면 `E-USE-ALIASED` 다. 별칭은 덧이름이 아니라 바꿔 부르기이고, 같은 모듈을 두 철자로

부르면 읽는 사람이 둘이 같은 것인지 확인해야 한다. 같은 이름을 두 번 들여오면 `E-NAME-COLLISION` 이 나는데, 그 진단이 권하는 해법이 이 별칭이다.

> **반례. 별칭을 적어 놓고 원래 이름으로 부른다**

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 ✘ 별칭을 적어 놓고 원래 이름으로 부른다 --- 같은 모듈을 두 철자로 부르는 코드가 된
다
  let p geom.point be make geom.point do x 7 . y 0 . end .
  return field p x .
end
```

실행 결과

```
$ 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.`
```

`as g` 를 적은 순간 이 파일에서 그 모듈의 이름은 `g` 하나다. `geom.point` 는 `E-USE-ALIASED` 로 거절된다. 두 철자가 다 서면 한쪽만 고친 코드가 생기고, 읽는 사람은 둘이 같은 모듈인지 매번 확인해야 한다. 원래 이름을 적는 자리는 `use` 줄 하나뿐이다.

21.4 검색 경로가 없다

많은 언어는 이름만 적으면 도구가 어딘가의 검색 경로에서 모듈을 찾아온다. 그러면 이 프로그램이 무엇에 기대는지가 소스 밖의 지식이 되고, 같은 소스가 기계에 따라 다른 파일을 가져올 수 있다. Lowent의 들여오기는 셋 가운데 하나다.

- 자리를 적는다: `use geom from "geom.low" .`
- 같은 번역 단위로 함께 넘긴다: `lowentc --check app.low geom.low`
- 표준 라이브러리의 예약된 이름을 쓴다: `use allocs .`

패키지 매니페스트(`pkg.low`)로 외부 의존을 적을 때도 원칙은 같다. 의존은 **내용 해시와 함께** 고정되고, 바이트가 바뀌면 빌드가 거절된다(31장).

> **흔한 오해. 모듈 이름은 파일 이름에서 온다**
>
> 표준 라이브러리에서 이 오해가 자주 걸린다. `use` 가 찾는 것은 파일 안의 `module` 선언이다. `lib/alloc.low` 의 모듈 이름은 `allocs` 이고, `lib/str.low` 는 `strings`, `lib/vec.low` 는 `vecs` 다. 이름이 다른 모듈의 목록은 표준 라이브러리 부(32장)의 첫 표에 있다.

21.5 이름이 부딪히면

한 모듈 안에서 같은 이름을 두 번 선언할 수 없다.

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
```

실행 결과

```
$ 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
```

두 번째가 첫 번째를 가리는 것이 아니라, 둘 가운데 하나에 조용히 닿을 수 없게 된다. 그래서 거절한다. 들여온 이름끼리 부딪히면 부르는 자리에서 모듈 이름으로 밝혀 적는다. 들여오기는 이미 있는 이름을 덮어쓰지 않는다(`E-NAME-COLLISION`).

21.6 최상위 선언의 차례는 무관하다

모듈 안의 최상위 선언은 차례와 상관이 없다. 뒤에 선언한 op 을 앞에서 부를 수 있고, 서로 부르는 op 도 된다.

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
```

실행 결과

```
$ lowentc --run is_even order.low 10
is_even(10) = 1
$ lowentc --run is_even order.low 7
is_even(7) = 0
```

`is_even` 은 뒤에 선언된 `is_odd` 를 부르고, `is_odd` 는 다시 `is_even` 을 부른다. 반면 op **본문 안**의 지역 이름은 쓰기 전에 선언해야 한다. 파일 전체는 훑어보며 읽지만 본문은 위에서 아래로 읽기 때문이다.

21.7 흔한 실수

반례. 들여온 이름을 모듈 이름 없이 부른다

examples/ch21/mistake_bare.low

```
module mistake_bare .
rem expect: E-VISIBILITY

use geom from "geom.low" .

fn origin_x output i64 .
do
  rem ✘ 들여온 이름을 모듈 이름 없이 불렀다
  let a point be make point do x 0 . y 0 . end .
  return field a x .
end
```

실행 결과

```
$ 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)
```

어떤 언어는 들여온 이름을 그냥 쓰게 하거나 `import *` 로 한꺼번에 풀어놓는다. 그러면 `point` 가 이 파일의 것인지 어느 모듈의 것인지 읽어서 알 수 없고, 들여오는 모듈이 늘면 이름이 조용히 부딪힌다. Lowent 에는 풀어놓기가 없다. 다른 모듈의 이름은 언제나 `geom.point` 처럼 한정해 적는다. 진단이 같은 줄에 두 번 나오는 것은 `let` 의 타입 자리와 `make` 자리를 따로 세기 때문이다.

반례. `from` 의 자리에 확장자를 빠뜨린다

examples/ch21/mistake_noext.low

```
module mistake_noext .
rem expect: E-DEP-MISSING

rem ✘ 자리는 파일 이름 그대로 --- `.low` 까지 적는다
use geom from "geom" .

fn same input v i64 . output i64 .
```

```
do
  return v .
end
```

실행 결과

```
$ 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
```

자리는 짐작되지 않는다. `"geom"` 이라고 적으면 도구는 정확히 `geom` 이라는 파일을 찾고, 없으면 `E-DEP-MISSING` 이다. 검색 경로가 없는 것과 같은 까닭이다 — 도구가 `.low` 를 붙여 보거나 다른 디렉터리를 뒤지기 시작하면 무엇에 기대는지가 다시 소스 밖으로 나간다. `use geom from "geom.low" .` 처럼 파일 이름을 그대로 적는다.

반례. 파일 이름으로 표준 모듈을 들여온다

examples/ch21/mistake_filename.low

```
module mistake_filename .
rem expect: E-IR-UNDEF

rem ✘ 파일은 `lib/str.low` 이지만 그 안의 모듈 이름은 `strings` 다
use str .

fn same input a slice u8 . input b slice u8 . output bool .
do
  return str.eq_str a b .
end
```

실행 결과

```
$ 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
```

`lib/str.low` 파일의 모듈 이름은 `strings` 다. `use str .` 은 그런 모듈을 찾지 못해 `W-USE-EXTERNAL` 로 "확인할 수 없다" 고 알리고, 이름을 실제로 쓰는 자리에서 `E-IR-UNDEF` 로 거절된다. 경고를 먼저 보고 모듈 이름을 확인한다.

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
```

실행 결과

```
$ 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]
```

반례. 내보낸 op 의 서명에 감춘 타입을 쓴다

examples/ch21/secretbox.low

```
module secretbox .
rem expect: W-EXPORT-HIDDEN

rem `export` 가 없는 타입 --- 이 모듈 밖에서는 이름을 적을 수 없다
struct secret do
  v u64 .
end

rem ✘ 내보낸 op 이 감춘 타입을 돌려준다 --- 들여온 쪽은 그 이름을 적을 수 없다
export fn make_secret output secret .
do
  return make secret do v 1 . end .
end
```

실행 결과

```
$ 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` 자체는 번역을 통과하지만 내보내는 쪽에서 `W-EXPORT-HIDDEN` 으로 알린다 — 이 export 는 밖에서 쓸 수 없다는 뜻이다. 그 알림이 없다면 문제는 들여오는 쪽에서야 드러난다.

examples/ch21/mistake_privtype.low

```
module mistake_privtype .
rem expect: E-VISIBILITY

use secretbox from "secretbox.low" .

fn peek output u64 .
do
  rem ✘ 받은 값을 담을 타입 이름이 감춰져 있다
  let s secretbox.secret be secretbox.make_secret .
```

```
  return field s v .
end
```

실행 결과

```
$ 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 은 내보냈지만 그 결과의 타입 secret 은 감췄다. 들여온 쪽은 결과를 담을 이름을 적을 수 없어 E-VISIBILITY 로 거절된다. 쓸 수 없는 export 인 셈이다. 내보낸 op 의 입력·출력 타입도 함께 export 한다.

반례. 다른 모듈의 열거형 갈래를 case 에 모듈 이름과 함께 적는다

examples/ch21/sizes.low

```
module sizes .

rem 다른 모듈이 쓸 열거형 — 갈래는 값을 지니지 않는다
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
```

실행 결과

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

sizes 모듈은 열거형 kind 와 그것을 돌려주는 classify 를 내보낸다. 들여온 쪽에서 갈래를 모듈 이름으로 한정해 적으면 이렇게 된다.

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 ✘ 모듈 이름을 붙인 갈래 — 이 판에서는 어떤 값에나 맞는다
    case sizes.small do
      return 1 .
    end
    case sizes.big do
      return 2 .
    end
  end
end
```

실행 결과

```
$ lowentc --run describe mistake_enumcase.low 5
describe(5) = 1
$ lowentc --run describe mistake_enumcase.low 500
describe(500) = 2
```

`describe 500` 은 `big` 이라 2 를, `describe 5` 는 1 을 돌려준다. 한정해 적어도 갈래로 읽히기 때문이다. 2026-09-16 까지는 그렇지 않았다 — 하강이 `case sizes.small` 을 갈래가 아니라 **아무 값에나 맞는 자리** 로 읽어 첫 갈래가 모든 값을 가져갔고, 500 도 1 을 냈다. 검사기는 한정 이름을 갈래로 좁히는데 하강은 그러지 않아 생긴 어긋남이었고, 이제 두 층이 같은 나무를 본다. 그래도 `case` 에는 갈래 이름만 적는 편이 짧고, 어느 열거형의 갈래인지는 `k` 의 타입이 정한다.

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 갈래 이름만 적는다 — 타입(sizes.kind)이 어느 갈래인지 정한다
    case small do
      return 1 .
    end
    case big do
      return 2 .
    end
  end
end
```

실행 결과

```
$ lowentc --run describe enumcase_fixed.low 5
describe(5) = 1
$ lowentc --run describe enumcase_fixed.low 500
describe(500) = 2
```

흔한 오해. 두 모듈이 서로를 들여오면 안 된다

examples/ch21/cycle_a.low

```
module cycle_a .
rem run: fa 3
rem run: fa 4

rem `cycle_b` 를 들여오고, `cycle_b` 도 이 모듈을 들여온다
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
```

실행 결과

```
$ lowentc --run fa cycle_a.low 3
fa(3) = 1
$ lowentc --run fa cycle_a.low 4
fa(4) = 0
```

`cycle_a` 는 `cycle_b` 를 들여오고 `cycle_b` 도 `cycle_a` 를 들여오지만 번역되고 돈다. C 의 헤더처럼 "먼저 읽은 것만 안다" 가 아니라, 번역 단위 전체를 모은 뒤 이름을 해소하기 때문이다. 최상위 선언의 차례가 무관한 것과 같은 원리다. 그래도 서로 들여오는 모듈은 하나로 합치거나 공통 부분을 셋째 모듈로 빼는 편이 읽기 쉽다. 되부름의 깊이는 계약(`requires le n 10 .`)이 막는다.

21.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`module geom .`	이 파일이 모듈 `geom` 이다(첫 줄)	모듈 이름은 파일 이름과 따로 — `use` 는 이 이름을 찾는다
`export fn manhattan …` · `export struct point …`	밖에서 보이게 한다	감춘 것이 기본 — 보이는 것은 약속
`use geom from "geom.low" .`	선언한 파일 기준의 자리에서 들여온다	검색 경로가 없다 — 무엇에 기대는지 소스에 보인다
`use allocs .`	같은 번역 단위나 표준 라이브러리에서 들여온다	없으면 `W-USE-EXTERNAL`
`geom.point` · `geom.manhattan a b`	들여온 이름은 모듈 이름으로 한정한다	풀어놓기가 없다
`geom.helper` **(감춘 이름)**	거절(`E-VISIBILITY`)	한정해도 문이 열리지 않는다
같은 이름을 두 번 선언 · 두 들여오기가 같은 이름	거절(`E-NAME-DUP` · `E-NAME-COLLISION`)	어느 쪽에 닿는지 조용히 정하지 않는다
최상위 선언의 차례	무관 — 서로 불러도 된다	파일은 훑어 읽고 본문은 위에서 아래로 읽는다

`use geom from "geom.low" as g .`	들여온 모듈을 이 파일에서 `g` 로 부른다	긴 이름 · 같은 이름을 풀어 준다 — 모듈의 실제 이름은 그대로
`export let …` • `export test …`	거절(`E-TOPLEVEL`)	상수는 값을 돌려주는 `fn` 으로 내보낸다
다른 모듈 열거형의 `case small`	갈래 이름만 적는다	`case sizes.small` 은 이 판에서 아무 값에나 맞는다

표 21.2 — 모듈의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

파일 하나가 모듈 하나이고, `export` 한 것만 밖에서 보인다. `use <모듈> from "<자리>" .` 는 선언한 파일 기준의 자리에서 들여오고, 자리를 적지 않으면 같은 번역 단위나 표준 라이브러리에서 찾는다. 검색 경로는 없다. 한정해도 감춘 이름에는 닿지 못하고, 같은 이름의 중복 선언은 거절된다. 최상위 선언의 차례는 무관하지만 본문의 지역은 먼저 선언한다.

22 제네릭 — 번역 시점에 정해지는 매개변수

이 장의 차례

22.1 값과 타입을 번역 시점에 받는다 261
22.2 조합마다 실물이 만들어진다 263
22.3 타입에 조건을 건다 263
22.4 값을 넘기지 않고 타입이 들고 온다 265
22.5 흔한 실수 266
22.6 이 장의 문법 한눈에 270

먼저 알아야 할 것

8장 식 · `size_of` 와 `comptime` 은 번역할 때 계산된다
20장 할당기와 고정 메모리 · 할당기를 `input comptime a type .` 으로 받아 갈아 끼운다

돌아보기

20장의 `two_from` 은 `bump_bytes` 와 `bump_aligned` 를 모두 받았다. 할당기를 갈아 끼우는 비용이 왜 0 이라고 했는가?

> **답.** 할당기의 **타입**이 번역할 때 구체 타입으로 정해져서, 호출마다 그 타입 전용 코드가 만들어지기 때문이다. 가상 함수 표도 간접 호출도 없다. 이 장은 그 "번역할 때 정해지는 매개변수" — `comptime` — 를 일반적으로 다룬다.

이 장의 필요성과 맥락

같은 알고리즘을 타입마다 베껴 쓰면 고칠 자리가 타입 수만큼 늘어난다. 함수 포인터로 비교를 넘기면 호출 자리에서는 공짜처럼 보이지만 간접 호출의 비용이 숨는다. Lowent 는 타입과 상수를 **번역 시점 매개변수**로 받고, 쓰인 조합마다 실물을 만든다. 무엇이 만들어지는지는 부르는 자리에 보인다. 트레이트(23장)와 `pipe`(24장)가 모두 이 위에 서므로, 추상의 도구로 먼저 세운다.

이 장이 끝나면

`input comptime n u8 .` 처럼 값을, `input comptime t type .` 처럼 타입을 번역 시점에 받는 법을 익힌다. 번역 시점에 알 수 없는 값을 주면 거절된다는 것, 쓰인 조합마다 실물이 만들어지고(단형화) 그 수가 곧 코드의 양이라는 것을 알게 된다. 타입 매개변수에 `requires <트레이트> t .` 로 조건을 걸고, 조건을 갖추지 못한 타입을 주면 무엇이 일어나는지도 보게 된다.

이 장에서 답할 질문

1. 타입 인자를 추론하지 않는 이유는 무엇인가? `max_of a b` 라고만 써도 `a` 의 타입에서 알 수 있지 않은가?

22.1 값과 타입을 번역 시점에 받는다

`comptime` 을 붙인 매개변수는 번역 시점에 값이 정해져야 한다. 타입도 그렇게 받는다.

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
```

실행 결과

```
$ lowentc --run report sizes.low
report() = 917
```

- `bytes_for` 의 `input comptime t type .` 은 타입을 받는다. 본문의 `size_of t` 는 그 타입의 크기를 번역할 때 준다. `bytes_for u8 100` 은 100, `bytes_for u64 100` 은 800 이다.
- `add_const` 의 `input comptime n u8 .` 은 값을 받는다. `add_const 7 10` 은 17 이다.
- 부르는 자리는 타입이나 상수를 **보통 인자처럼** 앞자리에 적는다. 꺾쇠(`<T>`)도, 추론도 없다.

comptime 매개변수는 머리에서 가장 앞(권한 입력보다도 앞)에 온다(3장). 뒤따르는 입력과 출력의 타입이 그 이름을 쓰기 때문이다.

번역 시점에 알 수 없는 값을 주면 거절된다.

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
```

실행 결과

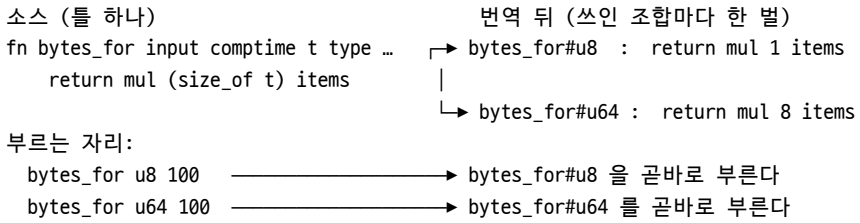
```
$ lowentc --check rt_arg.low
rt_arg.low:12:0 E-COMPTIME-ARG: this argument sits in a `comptime` parameter, so it must
 be a COMPILE-TIME CONSTANT (an integer literal, or a module `let`). A runtime value the
re makes the word `comptime` a lie — and the analysis, the folding and the monomorphisat
ion all believe it
```

`k` 는 실행할 때에야 정해진다. 그 자리에 `comptime` 이라고 적힌 이상, 실행 값을 받으면 그 낱말이 거짓말이 된다. comptime 자리에 올 수 있는 것은 정수 리터럴과 모듈 수준의 `let` 상수다.

> **문. 타입 인자를 추론하지 않는 이유는 무엇인가? `max_of a b` 라고만 써도 `a` 의 타입에서 알 수 있지 않은가?**
>
> **답.** 알 수 있다. 그러나 추론하면 **무엇이 만들어지는지**가 부르는 자리에서 보이지 않는다. 같은 줄이 인자의 타입에 따라 다른 실물을 부르고, 그 수는 소스를 따라가야 센다. Lowent 는 쓰인 조합의 수가 곧 만들어지는 코드의 양이라는 비용을 부르는 자리에서 셀 수 있게 두는 쪽을 골랐다.

22.2 조합마다 실물이 만들어진다

처리기는 쓰인 조합마다 그 조합 전용 op 을 만든다. 이것을 **단형화**(monomorphisation)라 한다. `bytes_for u8` 과 `bytes_for u64` 는 네이티브 코드에서 서로 다른 두 함수다. 타입이 확정된 뒤 만들어지므로 크기와 연산이 모두 상수로 박히고, 호출은 직접 호출이다.



틀에 적힌 `size_of t` 가 각 벌에서는 1 과 8 이라는 상수가 된다. 실행 중에 «t 가 무엇인가» 를 묻는 일이 남지 않는다.

대가는 코드의 양이다. 열 가지 타입으로 부르면 열 벌이 생긴다. 이 비용은 숨지 않는다. 부르는 자리에 타입이 적혀 있으니 몇 벌이 생길지를 소스에서 센다.

22.3 타입에 조건을 건다

타입을 받는 op 은 대개 그 타입이 무언가를 할 줄 알아야 한다. 최댓값을 고르려면 비교할 줄 알아야 한다. 그 조건을 `requires <트레이트> <타입> .` 으로 건다.

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
```

실행 결과

```
$ lowentc --run bigger bound.low 3 9
bigger(3, 9) = 9
$ lowentc --run bigger bound.low 12 5
bigger(12, 5) = 12
```

- `trait ordered` 는 "`less` 라는 op 을 갖춘다" 는 약속이다. `score` 는 `satisfies ordered .` 로 그 약속을 갖추겠다고 적고, `score.less` 로 실제로 갖춘다(23장).
- `max_of` 의 `requires ordered t .` 가 타입 조건이다. 본문은 그 조건을 믿고 `method a less b` 를 부른다.
- `bigger` 는 `max_of score …` 로 부른다. 처리기는 `max_of` 의 `score` 전용 실물을 만든다. 방출된 C 에서 그 함수의 이름에 `max_of_score` 가 들어 있는 것을 볼 수 있다.

조건을 갖추지 못한 타입으로 부르면 거절된다.

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
```

실행 결과

```
$ 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>`
```

타입 조건은 약속이고, 부르는 쪽이 그것을 넘어 실물을 만들게 하면 받는 쪽의 계약이 거짓이 된다. 진단은 그 타입에 `satisfies` 와 필요한 op 을 더하라고 한다.

> **흔한 오해. 제네릭 코드는 느리다**
>
> 언어에 따라 다르다. 타입을 지우고 모든 값을 상자에 담아 간접 호출로 다루는 방식은 느리다. 단형화하는 방식은 손으로 타입마다 베껴 쓴 코드와 같은 기계어가 된다. Lowent 는 단형화한다. 대가는 속도가 아니라 코드의 양이고, 그 양은 부르는 자리에서 셀 수 있다.

22.4 값을 넘기지 않고 타입이 들고 온다

C 의 `qsort` 는 비교 함수를 **값**으로 받는다. Lowent 에는 일급 함수가 없고, 있더라도 함수 포인터를 거치는 간접 호출이 숨는다. 대신 **비교를 타입이 들고 온다**. 표준 라이브러리 `sortgen` 의 머리가 그 모양이다.

```
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 .
```

정렬 기준을 바꾸려면 다른 `less` 를 가진 타입을 쓴다. 필드 하나짜리 구조체로 감싸도 배치는 그대로라서 비용이 없다. 내림차순이나 여러 키도 `less` 를 그렇게 쓰면 된다. 모드 인자를 다는 대신 타입이 뜻을 들고 온다(34장).

> **실제 사례. 할당 효과까지 타입이 정한다**
>
> 15장의 `via` 는 제네릭이 효과에 닿는 자리다. `vecgen.append` 는 `effects state via a .` 로 적혀 있어서, 빌린 바이트 위의 범프로 단형화하면 `state` 만, 힙에서 깎는 `heap_bytes` 로 단형화하면 `heap state` 가 그 인스턴스의 서명에 선다. 타입 인자가 알고리즘뿐 아니라 **효과**까지 정한다. 한 소스로 운영체제 없는 기계와 서버를 함께 다루는 방법이다.

22.5 흔한 실수

반례. 타입 인자를 빠뜨리고 추론되기를 기대한다

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 ✘ 앞자리의 타입 인자 `score` 를 빠뜨렸다 --- 추론은 없다
  let m score be max_of (make score do v x . end) (make score do v y . end) .
  return field m v .
end
```

실행 결과

```
$ 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)
```

`max_of` 의 첫 입력은 타입이다. 인자의 타입에서 `score` 를 알아낼 수 있어 보여도 Lowent 는 추론하지 않는다 — 무엇이 만들어지는지가 부르는 자리에 보여야 하기 때문이다. 앞자리를 비우면 `E-MONO-NOTYPE` 으로, 타입을 앞에 적으라고 말한다. 전에는 같은 자리가 "`max_of` 라는 이름이 없다"(`E-IR-UNDEF`)였다 — op 은 분명히 있으므로 틀린 안내였다.

반례. 본문이 쓰는 행동을 타입 조건으로 적지 않는다

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 ✘ 본문은 `less` 를 부르는데 머리에 `requires ordered t .` 가 없다
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
```

실행 결과

```
$ 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
```

`max_of` 의 본문은 `less` 를 부르지만 머리에 `requires ordered t .` 가 없다. 그러면 `plain` 으로 부른 잘못이 **템플릿 안**의 줄에서 `E-METHOD-UNDEF` 로 나고, 도구는 `N-MONO-SITE` 로 "그 인스턴스를 청한 줄은 여기" 라고 덧붙인다. 조건을 적은 `unsat.low` 는 같은 잘못을 부르는 자리에서 `E-BOUND-UNSAT` 으로 곧바로 말한다. 타입 조건은 부르는 쪽에게 주는 약속이면서 진단을 제자리로 데려오는 표시다.

반례. 타입 자리에 크기를 넘긴다

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 ✘ 타입 자리에 크기(8)를 넘겼다 --- `u64` 라고 적는다
  return bytes_for 8 100 .
end
```

실행 결과

```
$ 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` 가 8 바이트이니 8 을 넘기면 될 것 같지만, `input comptime t type .` 이 받는 것은 **타입**이다. 이 판의 도구는 수 8 을 타입으로 읽으려다 템플릿 안의 `size_of t` 에서 `E-IR-UNDEF` 를 낸다. `bytes_for u64 100` 처럼 타입 이름을 적는다. 크기가 필요한 것은 템플릿의 일이고, 부르는 쪽은 무엇의 크기인지를 말한다.

반례. `comptime` 입력을 자료 입력 뒤에 둔다

examples/ch22/mistake_comptimeorder.low

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

rem ✘ `comptime` 입력이 자료 입력 뒤에 왔다
fn add_const input a u8 . input comptime n u8 . output u8 .
  requires le a 200 .
do
  return add a n .
end
```

실행 결과

```
$ 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)
```

머리의 차례는 하나로 정해져 있다. `comptime` 입력은 권한 입력보다도 앞, 가장 앞이다. 뒤따르는 입력과 출력의 타입이 그 이름을 쓸 수 있어야 하고, 부르는 자리에서도 "무엇을 만들지" 가 먼저 읽혀야 하기 때문이다. `E-CLAUSE-ORDER` 의 진단은 머리 차례 전체를 한 줄로 보여 준다.

흔한 오해. 값이 늘 같은 지역 `let` 은 번역 시점 상수다

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 ✘ 값은 늘 7 이지만 op 안의 `let` 은 실행 중에 생기는 이름이다
  let k u8 be 7 .
  return add_const k 10 .
end
```

실행 결과

```
$ 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` 는 언제나 7 이지만, op 안의 `let` 은 op 이 불릴 때 생기는 이름이다. `comptime` 자리가 받는 것은 정수 리터럴과 **모듈 수준의** `let` 뿐이다. 식을 따라가 "결국 상수" 인지 판정하기 시작하면, 어느 식이 번역 시점에 풀리는지가 도구의 영리함에 달린다. 상수로 쓸 값은 모듈 수준에 이름을 붙인다.

examples/ch22/module_const.low

```
module module_const .
rem run: seventeen

rem 모듈 수준의 `let` 은 번역 시점 상수다
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
```

실행 결과

```
$ lowentc --run seventeen module_const.low
seventeen() = 17
```

22.6 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`input comptime t type .`	타입을 번역 시점에 받는다(머리맨 앞)	뒤따르는 타입이 그 이름을 쓴다
`input comptime n u8 .`	값을 번역 시점에 받는다	상수로 접히고 검사가 지워진다
`bytes_for u64 100 · add_const 7 10`	타입과 상수를 앞자리 인자로 적는다	꺾쇠도 추론도 없다 — 무엇이 만들어지는지 보인다
`let step u8 be 7 .`**(모듈 수준)**	`comptime` 자리에 올 수 있는 이름 붙은 상수	op 안의 `let` 은 실행 중의 이름
`size_of t`	번역 시점에 타입의 크기	크기가 상수로 박힌다
`requires ordered t .`	타입 조건 — 트레이트를 갖추어야 한다	못 갖추면 부르는 자리에서 `E-BOUND-UNSAT`
`method a less b`	조건이 약속한 op 을 부른다	직접 호출로 단형화된다
쓰인 조합마다 실물 한 벌	단형화	비용은 속도가 아니라 코드의 양 — 소스에서 센다

표 22.1 — 제네릭의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`comptime` 매개변수는 번역 시점에 값이 정해지고, 타입도 `input comptime t type .` 으로 받는다. 부르는 자리는 타입과 상수를 앞자리 인자로 적으며 추론은 없다. 실행 값을 comptime 자리에 주면 거절된다. 쓰인 조합마다 실물이 만들어져 직접 호출이 되고, 그 수가 곧 코드의 양이다. `requires <트레이트> t .` 는 타입 조건이고, 갖추지 못한 타입은 거절된다. 비교 같은 행동은 값이 아니라 타입이 들고 온다.

23 트레이트 — 여러 타입이 지키는 한 가지 약속

이 장의 차례

23.1 무엇에 쓰나 272
23.2 타입에 붙은 op 과 `method` 273
23.3 op 을 여럿 가진 트레이트 273
23.4 서명에는 `fn·proc` 을 적지 않는다 275
23.5 갖추지 못했을 때 277
23.6 `via self` — 할당 효과만 더 적을 수 있다 278
23.7 흔한 실수 279
23.8 이 장의 문법 한눈에 283

먼저 알아야 할 것

10장 묶음 · `struct` 와 `field`
15장 효과 · 효과 줄은 좁히는 절이고 부르는 쪽으로 번진다
22장 제네릭 · `requires <트레이트> t .` 는 타입 조건이다

돌아보기

22장의 `max_of` 는 `requires ordered t .` 를 믿고 본문에서 무엇을 불렀는가? 그리고 조건을 갖추지 못한 `plain` 을 주면 어떻게 되었는가?

> **답.** `method a less b` 를 불렀다. `plain` 을 주면 `E-BOUND-UNSAT` 으로 거절되었다. 조건은 약속이고, 약속을 넘어 실물을 만들면 받는 쪽의 계약이 거짓이 되기 때문이다. 이 장은 그 약속 — 트레이트 — 를 선언하고 갖추는 법 전체를 다룬다.

이 장의 필요성과 맥락

사각형과 정사각형은 넓이를 구하는 방법이 다르지만 "넓이를 알려 준다" 는 약속은 같다. 그 약속에 이름을 붙이면, "넓이를 알려 주는 것이면 무엇이든" 받는 op 을 한 번만 쓸 수 있다. 이 언어에는 상속이 없으므로 여러 타입을 한 이름으로 다루는 길은 트레이트다. 할당기(20장)와 정렬(22장)에서 이미 쓴 도구를 여기서 제대로 편다.

이 장이 끝나면

타입에 op 을 붙이고(`fn rect.area`) `method` 로 부르는 법을 익힌다. 트레이트를 선언하고 `satisfies` 로 갖추며, op 을 여럿 가진 트레이트를 쓰는 법을 알게 된다. 트레이트 서명에 `fn·proc` 을 적지 않고 `effects` 줄이 갖추는 쪽을 정한다는 규칙, 갖추지 못했을 때의 진단들, `via self` 와 트레이트가 상속이 아니라는 점도 보게 된다.

이 장에서 답할 질문

1 트레이트의 op 에 기본 구현을 줄 수 있는가?

23.1 무엇에 쓰나

examples/ch23/why.low

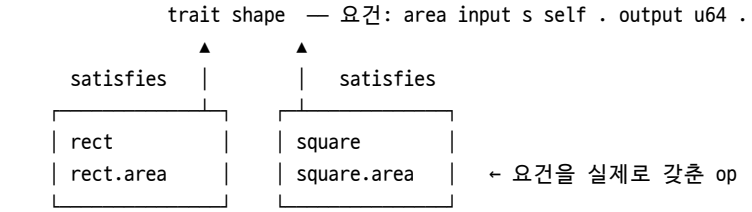
```
module why .
rem run: demo

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
```

실행 결과

```
$ lowentc --run demo why.low
demo() = 44
```

- `trait shape do … end` 는 "area 를 갖춘다" 는 약속이다. 서명의 `self` 는 이 약속을 갖출 타입 자신이다.
- `rect` 와 `square` 는 몸 안에 `satisfies shape .` 를 적어 약속을 갖추겠다고 선언한다.
- `fn rect.area` 와 `fn square.area` 가 실제로 갖춘다. 이름의 `rect.` 는 그 op 이 `rect` 에 붙었다는 뜻이다.
- `double_area` 는 **어떤 모양이든** 받는다. `requires shape t .` 가 "넓이를 알려 주는 타입만" 이라고 못 박는다.
- `method s area` 는 `s` 의 타입에 붙은 `area` 를 부른다. `rect` 면 `rect.area`, `square` 면 `square.area` 다.

트레이트는 **자격 요건**에 빗대면 쉽다. «넓이를 알려 줄 수 있음» 이라는 요건이 있고, 요건을 갖춘 타입만 그 요건을 요구하는 op 에 들어올 수 있다.




```
double_area 는 requires shape t .  — shape 를 갖춘 타입만 받는다
  double_area rect r    →  method s area  =  rect.area
  double_area square q  →  method s area  =  square.area
```

`double_area rect r` 과 `double_area square q` 는 각각 전용 실물로 단형화되므로(22장), `method` 는 실행 중에 찾는 것이 아니라 번역할 때 정해진다. 가상 함수 표가 없다.

23.2 타입에 붙은 op 과 `method`

트레이트와 상관없이도 op 을 타입에 붙일 수 있다. 이름 앞에 타입 이름과 점을 붙이면 되고, 첫 입력은 그 타입의 값이다. `method <값> <이름> <인자…>` 가 그 op 을 부른다. 수신자가 다른 폼의 결과여도 된다 — `method (method r grow 1) area`. 붙은 op 이 없으면 거절된다.

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
```

실행 결과

```
$ 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)
```

없는 이름을 부르면 위로 찾아 올라가는 일이 없다. 붙은 op 은 이름 칸을 나누는 장치이지 상속이 아니다.

23.3 op 을 여럿 가진 트레이트

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
```

실행 결과

```
$ 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)
```

적는 법은 셋이다.

- 서명마다 한 줄이다. 줄은 op 의 **이름**으로 시작하고, op 머리와 같은 차례로 절을 적는다 — 입력, 출력, 효과. 다음 이름이 나오면 다음 서명이다. 몇 개든 적는다.
- `grow` 처럼 출력이 `self` 면 갖추는 쪽은 자기 타입(`rect`)을 돌려준다.
- 갖추는 op 의 이름은 `<타입>.<이름>` 이다.

`demo_empty` 는 폭이 0 인 사각형으로 `checked_area` 를 불러 멈춘다. 서명이 `effects panic` 이라고 약속했으므로 부르는 쪽은 멈출 수 있다는 것을 안다.

> **실제 사례. `method` 로 부른 효과가 번지지 않는 결함**
>
> 위 예제를 `--check` 하면 `demo` 에 `W-EFFECT-OVER`(`panic` 을 선언했지만 하지 않는다)라는 경고가 붙는다. 틀린 경고다. `demo_empty` 는 실제로 멈춘다. 이 책을 쓰며 확인한 바로, 이 판의 처리기는 `method` 로 부른 op 의 효과를 부르는 쪽에 번지게 하지 않는다. 그래서 순수한 `fn` 이 `method` 로 `panic` 하는 op 을 불러도 번역이 통과한다. 같은 op 을 `rect.checked_area r` 로 직접 부르면 올바르게 거절된다. 명세(15장)가 요구하는 동작은 직접 부름 쪽이고, `method` 쪽은 결함이다. 고쳐지기 전까지는 효과가 있는 붙은 op 을 순수한 op 안에서 `method` 로 부르지 않는다.

23.4 서명에는 `fn`·`proc` 을 적지 않는다

트레이트 서명에 `fn` 이나 `proc` 을 적으면 거절된다.

examples/ch23/sig_kind.low

```
module sig_kind .
rem expect: E-TRAIT-SIG

trait shape do
  fn area input s self . output u64 .
end
```

실행 결과

```
$ 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)
```

그 op 이 무엇을 할 수 있는지는 서명의 `effects` 줄이 정한다.

서명의 `effects`	갖추는 쪽
없음	`fn`, 또는 `effects` 를 적은 `proc`
`effects panic` 처럼 있음	그 효과나 그보다 적은 효과를 적은 `proc`. 효과를 안 쓰면 `fn` 도 된다

표 23.1 — 서명의 `effects` 와 갖추는 쪽

`effects` 줄이 **없는** `proc` 은 좁히지 않은 것이라 무엇이든 할 수 있다고 읽힌다(5장). 그래서 효과 없는 서명을 그런 `proc` 으로 갖추면 거절된다.

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
```

실행 결과

```
$ 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
```

서명보다 많은 효과를 적어도 거절된다.

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
```

실행 결과

```
$ 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)
```

부르는 쪽은 **트레이트의 계약**을 보고 판단한다. 갖춘 쪽이 계약에 없는 일을 하면 그 판단이 틀린다. 이름만 맞고 효과가 다르면 트레이트는 계약을 실어 나르지 못한다.

23.5 갖추지 못했을 때

진단	뜻
E-TRAIT-UNDEF	satisfies 가 선언되지 않은 트레이트를 부른다
E-TRAIT-MISSING	목록의 op 하나가 없다
E-TRAIT-SIG	서명에 fn·proc 을 적었거나, op 은 있는데 매개변수 수가 다르다
E-TRAIT-EFFECT	갖춘 op 이 서명보다 많은 효과를 가진다
E-TRAIT-RECV	타입이 아니라 op 에 satisfies 를 적었다

표 23.2 — 트레이트를 갖추지 못한 자리

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
```

실행 결과

```
$ 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 는 주석이 아니다. 갖추겠다고 적은 순간부터 처리기가 목록 전체를 검사한다.

> **문. 트레이트의 op 에 기본 구현을 줄 수 있는가?**
>
> **답.** 줄 수 없다. 트레이트는 목록일 뿐 코드를 담지 않는다. 여러 타입이 같은 구현을 나누고 싶다면, 그 구현을 제네릭 op (double_area 처럼)으로 쓰고 타입들은 그 op 이 필요로 하는 최소한만 갖춘다. 기본

> 구현이 있으면 어떤 타입의 area 가 어디서 왔는지를 찾아 올라가야 하고, 그것이 상속이 만드는 엔트로피다.

23.6 via self — 할당 효과만 더 적을 수 있다

할당기 트레이트처럼 구현마다 할당 효과가 다른 약속이 있다. 서명의 효과 줄에 via self 를 적으면, 갖추는 쪽은 할당 계열 효과(alloc·heap·lock·atomic)를 서명보다 **더** 적을 수 있다. 그 밖의 효과를 더 적는 것은 여전히 E-TRAIT-EFFECT 다.

```
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
```

범프의 reserve 는 state 뿐이고, 힙에서 깎는 heap_bytes 의 reserve 는 heap state 다. 그 차이가 제네릭 op 의 via a 를 따라 부르는 쪽까지 올라간다(20장).

할당기가 아닌 약속에도 같은 장치를 쓴다. 번호표를 내주는 트레이트를 두 액터가 갖추는데, 하나는 수만 세고 하나는 번호표마다 고정 창에서 바이트를 깎는다.

examples/ch23/viaself.low

```
module viaself .
rem run: main

rem 번호표를 내주는 약속. 구현이 할당 계열 효과를 더 적어도 된다
trait ticketer do
  issue input s self . output u64 . effects state via self .
end

rem 수를 세기만 한다 — 효과는 state 뿐
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 번호표마다 고정 창에서 8 바이트를 깎는다 — alloc 을 더 적는다
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 어느 구현이든 받는다 — via t 가 그 구현의 효과를 여기로 나른다
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
```

실행 결과

```
$ lowentc --run main viaself.low
main() = 33
```

- `issue input s self . output u64 . effects state via self .` 이 서명이다. `counter` 는 `effects state .` 로, `carver` 는 `effects alloc state .` 로 갖춘다. `alloc` 은 할당 계열이라 `via self` 가 허락한다.
- `issue_two` 는 `effects state via t .` 라고 적었다. `counter` 로 단형화한 인스턴스의 효과는 `state` 이고, `carver` 로 단형화한 인스턴스의 효과는 `alloc state` 다. 효과가 **인스턴스마다** 정확하다.
- 그래서 `main` 은 `cap allocator` 를 받고 `alloc` 을 적어야 한다. `carver` 로 부르는 줄이 없다면 `main` 에 `alloc` 이 필요 없다.
- 답 33 은 `counter` 의 1 + 2 와 `carver` 의 10 + 20 이다.

`carver` 의 상태에는 `root cap allocator .` 라는 **권한 칸**이 있다. 그 칸의 규칙은 25장에서 다룬다.

> **흔한 오해. 트레이트를 갖추면 그 트레이트의 무언가를 물려받는다**
>
> 물려받는 것은 없다. 트레이트를 갖췄다는 것은 "그 op 들을 갖고 있다" 는 **사실**일 뿐이고, 타입 사이에 위아래가 생기지 않는다. `rect` 와 `square` 는 `shape` 를 갖춘 뒤에도 서로 아무 관계가 없다. 이 언어에는 상속이 없다.

23.7 흔한 실수

> **반례. `method` 뒤에 이름을 먼저 적는다**
>
> `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 ✘ `method` 뒤에는 값이 먼저, 이름이 다음이다
  return method area r .
end
```

실행 결과

```
$ 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)
```

객체 지향 언어의 `r.area()` 를 거꾸로 옮기면 `method area r` 이 되기 쉽다. `method` 는 **값을 먼저** 받는다 — 어느 타입에 붙은 op 을 찾을지 값의 타입이 정하기 때문이다. 이름을 먼저 적으면 도구는 `area` 를 값으로 읽고 그 타입에 붙은 `r` 을 찾다가 `E-METHOD-UNDEF` 를 낸다. `method r area` 로 적는다.

반례. 갖추는 op 의 이름에 타입 접두사를 빠뜨린다

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 ✘ 이름에 `rect.` 이 없다 --- 타입에 붙지 않은 보통 op 이다
fn area input s rect . output u64 .
do
  return mul (field s w) (field s h) .
end
```

실행 결과

```
$ 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 .` 는 `rect` 를 받는 **보통 op** 이지 `rect` 에 붙은 op 이 아니다. 트레이트는 `rect.area` 를 찾으므로 `E-TRAIT-MISSING` 이다. 받는 타입만 보고 붙여 주지 않는 까닭은, 같은 타입을 받는 보통 op 이 여럿일 때 어느 것이 약속을 갖춘 것인지 이름만 보고 알게 하려는 것이다. 이 진단에는 파일 이름이 빠져 있다. 줄 번호는 `satisfies` 를 적은 타입 선언을 가리킨다.

반례. op 만 있으면 트레이트를 갖춘 것으로 여긴다

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` 는 있지만 `satisfies shape .` 를 적지 않았다
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
```

실행 결과

```
$ 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>`
```

`rect.area` 가 있으니 "넓이를 알려 주는 타입" 인 것은 맞다. 그래도 `satisfies shape .` 를 적지 않으면 `double_area rect r` 은 `E-BOUND-UNSAT` 이다. 모양만 맞으면 통과시키는 방식(덕 타이핑)에서는 우연히 이름이 같은 op 이 약속을 갖춘 것으로 잘못 읽힌다. `satisfies` 는 "나는 이 계약을 지키겠다" 는 선언이고, 그 선언이 있어야 처리기가 목록 전체를 검사한다.

반례. 수신자를 첫 입력이 아닌 자리에 둔다

examples/ch23/mistake_recvlast.low

```
module mistake_recvlast .
rem expect: E-METHOD-RECV
```

```
struct rect do
  w u64 .
  h u64 .
end
rem ✘ 수신자 `s` 가 첫 입력이 아니다
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
```

실행 결과

```
$ 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
```

붙은 op 의 첫 입력은 수신자다. `method r scaled 5` 는 `r` 을 첫 자리에, 5 를 둘째 자리에 넣는다. `scaled` 는 첫 자리를 `k`, 둘째를 `s` 로 받았으므로 구조체에 곱셈을 하려다 멈춘다. 그런 머리는 `E-METHOD-RECV` 로 번역 때 거절한다. 붙은 op 은 `input s rect .` 를 언제나 맨 앞에 둔다.

흔한 오해. 타입에 붙은 op 은 `method` 로만 부를 수 있다

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 같은 op 을 두 길로 부른다 --- 이름으로 곧장, 그리고 `method` 로
  return add (rect.area r) (method r area) .
end
```

실행 결과

```
$ lowentc --run f direct_call.low
f() = 12
```

> `rect.area r` 은 이름으로 곧장 부르고, `method r area` 는 값의 타입을 보고 같은 op 을 찾아 부른다. 둘은 같은 op 이고 결과도 6 과 6 이다. `method` 가 쓸모 있는 곳은 타입이 제네릭 매개변수여서 이름을 적을 수 없는 자리(`double_area` 의 `method s area`)다. 타입을 아는 자리에서는 직접 부르는 편이 효과도 올바르게 번진다(위의 결함 사례).

23.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`fn rect.area input s rect . …`	타입에 op 을 붙인다 — 첫 입력이 수신자	이름 칸을 나누는 장치이지 상속이 아니다
`method r area • rect.area r`	값의 타입으로 찾아 부른다 · 이름으로 곧장 부른다	번역 때 정해진다 — 가상 함수 표가 없다
`trait shape do area input s self . output u64 . end`	타입이 갖출 op 의 목록	`self` 는 갖출 타입 자신
서명 줄: 이름 · 입력 · 출력 · 효과	op 머리와 같은 차례, `fn·proc` 은 적지 않는다	효과 줄이 갖추는 쪽의 상한
`struct rect do satisfies shape . … end`	이 타입이 약속을 갖춘다고 선언	선언이 있어야 목록 전체를 검사한다
`requires shape t .`	제네릭 op 의 타입 조건	갖추지 못한 타입은 `E-BOUND-UNSAT`
`effects state via self .`**(서명)**	할당 계열 효과만 더 적어도 된다	할당기마다 효과가 다르다
`E-TRAIT-MISSING` • `-SIG` • `-EFFECT` • `-RECV` • `-UNDEF`	갖추지 못한 자리마다의 진단	`satisfies` 는 주석이 아니다
`effects state via t .`**(제네릭 op)**	단형화한 타입이 `via self` 로 더 적은 효과를 물려받는다	`counter` 로는 `state`, `carver` 로는 `alloc state`

표 23.3 — 트레이트의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

> `fn <타입>.<이름>` 은 타입에 op 을 붙이고 `method` 가 부른다. 트레이트는 타입이 갖출 op 의 목록이고, 타입은 몸 안의 `satisfies` 로 갖추겠다고 선언한다. 서명은 이름으로 시작해 입력·출력·효과 차례로 적고, `fn·proc` 은 적지 않는다. 서명의 `effects` 가 갖추는 쪽의 효과 상한이다. 어긋나면 `E-TRAIT-*` 로 거절되고, `via self` 는 할당 계열 효과만 더 적게 허락한다. 트레이트는 상속이 아니다.

24 pipe — 하려는 일만 한 줄씩

이 장의 차례

24.1 같은 일, 두 가지 모양 285
24.2 스테이지와 종결자 286
24.3 한 번에 훑는 것은 정의다 290
24.4 필요한 만큼만 읽는다 290
24.5 이름이 같은 내장 map·filter 291
24.6 흔한 실수 292
24.7 이 장의 문법 한눈에 295

먼저 알아야 할 것

7장 흐름 · for 는 슬라이스를 훑고 세는 반복은 while 로 적는다
9장 줄 · mut slice 여야 원소에 쓸 수 있다
22장 제네릭 · 값 대신 이름 붙은 op 을 건넨다

돌아보기

7장에서 "for 로 0 부터 n 까지 세는 반복은 어떻게 적는가" 라는 물음에, 슬라이스를 걸러 세거나 모으는 일은 무엇이 맡는다고 했는가?

> **답.** pipe 가 맡는다고 했다. 세는 반복은 while 과 var 로 적고, 슬라이스를 거르고 바꾸고 모으는 일은 pipe 로 적는다. 이 장이 그 pipe 를 다룬다.

이 장의 필요성과 맥락

슬라이스를 훑는 반복은 거의 언제나 같은 뼈대다. 색인을 두고, 끝인지 보고, 원소를 꺼내 조건을 보고, 무언가를 쌓고, 색인을 늘린다. 이 뼈대를 손으로 쓰면 **하려는 일**("숫자만 남기고 센다")이 색인과 카운터 사이에 묻히고, 경계를 한 칸 틀리는 실수가 거기서 난다. pipe 는 뼈대를 언어가 맡고 사람은 하려는 일만 줄마다 한 낱말로 적게 한다. 그러면서도 손으로 쓴 반복과 똑같이 한 번만 훑는다. 추상(제6부)의 마지막 장으로, 추상이 비용을 숨기지 않는 모양을 보인다.

이 장이 끝나면

pipe <원천> do <스테이지…> <종결자> end 의 모양과, 스테이지 일곱·종결자 다섯을 알게 된다. 스테이지에 이름 붙은 op 을 건네는 법, collect into 로 호출자의 버퍼에 담는 법, fold·any·all·take 가 필요한 만큼만 읽는다는 것을 익힌다. 한 pipe 가 중간 배열 없이 한 번에 훑는 것이 최적화가 아니라 **정의**라는 점도 이해하게 된다.

이 장에서 답할 질문

1 collect into 로 쓰는 op 인데 머리가 effects none 이다. 호출자의 버퍼를 바꾸는데 효과가 없는가?

24.1 같은 일, 두 가지 모양

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
```

실행 결과

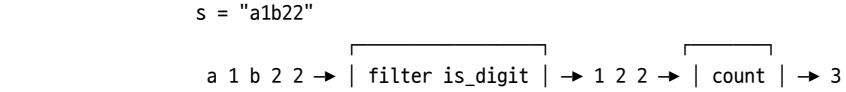
```
$ lowentc --run digits_loop digits.low [97,49,50,98,51]
digits_loop([97,49,50,98,51]) = 3
  arg0 (written) = [97,49,50,98,51]
$ lowentc --run digits_pipe digits.low [97,49,50,98,51]
digits_pipe([97,49,50,98,51]) = 3
  arg0 (written) = [97,49,50,98,51]
```

두 op 은 같은 답을 낸다. `digits_loop` 는 카운터 `n`, 색인 `i`, 끝 조건, 증가를 사람이 맞춘다. `digits_pipe` 는 두 줄이다.

- `filter is_digit .` — `is_digit` 이 참인 원소만 남긴다.
- `count .` — 남은 원소를 센다. 흐름을 끝내는 낱말(**종결자**)은 정확히 하나다.

`pipe` 는 문장이지만 값을 내는 종결자(`count`·`fold`·`any`·`all`)로 끝나면 식처럼 `return pipe s do … end .` 로 쓸 수 있다 — 끝의 점은 `return` 문장의 것이다(`end` 는 `pipe` 블록만 닫는다).

원소가 줄지어 흘러가는 관으로 그리면 이렇다.



```
└──────────────┘          └──────┘
  스테이지: 거른다          종결자: 흐름을 끝내고 값을 낸다
```

원소는 하나씩 관을 지나간다. 중간에 «숫자만 모은 배열» 같은 것이 따로 만들어지지 않는다.

24.2 스테이지와 종결자

낱말	op 의 모양	하는 일	끝·빈 흐름
`filter <op>`	`t → bool`	op 이 참인 원소만 보낸다	—
`map <op>`	`t → u`	원소마다 op 의 답을 보낸다. 타입이 `u` 로 바뀔 수 있다	—
`take <n>`	—	앞의 `n` 개만 보낸다	`n` 개 뒤 **훑기를 멈춘다**
`skip <n>`	—	앞의 `n` 개를 버린다	—
`enumerate <op>`	`(u64, t) → u`	순번(0 부터)과 원소를 op 에 준다	—
`zip <슬라이스> <op>`	`(t, t2) → u`	원소와 다른 슬라이스의 같은 자리 원소를 op 에 준다	짧은 쪽이 끝나면 끝난다
`scan <초기값> <op>`	`(a, t) → a`	누산한 **뒤의** 값을 원소로 보낸다 (초기값은 안 보낸다)	—
`collect into <버퍼>`	—	**종결자.** 버퍼에 앞에서부터 담고 **담은 개수**(`u64`)를 낸다	버퍼가 모자라면 길이를 알 때 번역이 거절(`E-COLLECT-FULL`), 모를 때 실행 중 멈춤
`fold <초기값> <op>`	`(a, t) → a`	**종결자.** 누산한 마지막 값(`a`)을 낸다	비면 초기값
`count`	—	**종결자.** 원소 수(`u64`)를 낸다	비면 0
`any <op>`	`t → bool`	**종결자.** 참인 원소가 있는가	첫 참에서 멈춘다 · 비면 거짓
`all <op>`	`t → bool`	**종결자.** 모두 참인가	첫 거짓에서 멈춘다 · 비면 참

표 24.1 — `pipe` 의 낱말 — 원소 타입 `t`, 누산값 타입 `a`

`filter·any·all` 에 주는 op 은 **판정**이라 `bool` 을 내야 한다. 수를 내는 op 을 주면 `E-PIPE-PRED` 로 거절된다 — 0 아닌 수를 참으로 읽는 규칙은 이 언어에 없다(13장의 `cast` 와 같은 까닭).

`pipe` 를 `return pipe … end` . 처럼 값 자리에 쓰면 그 값은 종결자가 낸 값이다. 스테이지는 줄마다 하나씩 원소를 다음 줄로 넘기고, 종결자가 마지막에 받아 값 하나로 만든다.

```
xs = [1, 5, 2, 7]
     |
     ▼  filter big      (big = gt x 2)       5, 7          1·2 는 여기서 버린다
     ▼  map dbl         (dbl = x + x)        10, 14
     ▼  collect into out                     out = [10, 14, …]   답 = 2 (담은 개수)
```

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
```

실행 결과

```
$ 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` 는 2 보다 큰 것만 남겨(`filter`) 두 배로 만들고(`map`) `out` 에 담는다(`collect into`). 결과 `[6,8,10,0,0]` 에서 담기지 않은 자리는 그대로다. 담을 버퍼는 **부르는 쪽이 준다.** `pipe` 는 할당하지 않는다.
- `sum_big` 의 `fold 0 addu` 는 0 에서 시작해 `addu` 로 누적한다.
- `has_zero` 는 `any is_zero`, `all_small` 은 `all under10` 이다.
- `middle` 은 `skip 1` 과 `take 2` 로 가운데 두 원소를 담는다.

스테이지에 주는 것은 **이름 붙은 op** 이다. 이 언어에는 이름 없는 함수(람다)가 없다. 스테이지 op 은 원소 하나를 받고 권한을 받지 않는다. `fold`·`scan` 의 op 은 누산값과 원소 둘을 받는다.

튜플이 없는데 순번이나 짝은 어떻게 다루는가? 쌍을 **만들지 않고 op 에 인자로 건넨다**.

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
```

실행 결과

```
$ 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` 는 `idxadd` 에 순번과 원소를 준다. `zip ys addb` 는 `xs` 의 원소와 `ys` 의 같은 자리 원소를 `addb` 에 준다. 짧은 쪽이 끝나면 흐름 전체가 끝나서 짝이 늘 맞는다. `scan 0 addb` 는 누적합을 원소로 흘린다. 쌍을 만들었다 곧장 풀지 않으므로 숨은 할당도 없다.

> **문. `collect into` 로 쓰는 op 인데 머리가 `effects none` 이다. 호출자의 버퍼를 바꾸는데 효과가 없는가?**
>
> **답.** 명세의 예제도 `effects none` 으로 적는다. `collect into` 는 `effects state` 를 **요구하지는** 않지만, 적어 두면 처리기가 그것을 호출자 저장소 쓰기로 세므로 "선언했지만 하지 않는다" 는 경고는 나지 않는다 — `mut`·`mut_ref` 매개변수에 쓰는 자리와 같은 판정이다. 어느 쪽으로 적든 호출자는 `mut slice` 를 건네는 자리에서 쓰기가 일어날 수 있다는 것을 안다(12장).

24.3 한 번에 훑는 것은 정의다

한 `pipe` 는 **한 번의 훑기**로 실행되고 스테이지 사이에 중간 배열이 만들어지지 않는다. 이것은 처리기가 해 볼 수 있는 최적화가 아니라 `pipe` 의 정의이고, 중간 배열을 만드는 처리기는 적합하지 않다.

왜 정의로 두는가. 융합을 최적화로 두면 "어디까지 융합되는가" 가 처리기마다 다르고, 쓰는 사람은 **절벽**을 만난다. 한 줄을 고쳤더니 갑자기 중간 배열이 생기고 느려지는 자리다. 그 절벽은 소스만 봐서는 안 보인다. Lowent 는 융합할 수 없는 연산을 스테이지 낱말에 **아예 넣지 않았다**. 정렬처럼 전체를 봐야 하는 연산이 스테이지에 없는 이유다.

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
```

실행 결과

```
$ 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
```

스테이지 목록은 닫혀 있다. `pipe` 가 식이 아니라 문장인 것도 같은 이유다. 식이었다면 스테이지를 값으로 떼어 넘길 수 있고, 그러면 어디까지가 한 줄기인지 소스에서 안 보인다. `do … end` 가 그 경계를 눈에 보이게 한다.

24.4 필요한 만큼만 읽는다

`any` 는 참을 내는 첫 원소에서, `all` 은 거짓을 내는 첫 원소에서, `take n` 은 `n` 개를 지나보낸 뒤 훑기를 멈춘다. 멈춘 뒤의 원소는 읽히지 않고 그 원소에 대한 스테이지도 실행되지 않는다. `has_zero [4,0,9]` 는 9 를 보지 않는다. 이 성질이 의미에 적혀 있어야 끝이 없는 원천도 `pipe` 로 다룰 수 있다.

종결자는 흐름을 끝낸다. 종결자 뒤에 스테이지가 오면 거절된다.

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
```

실행 결과

```
$ 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)
```

> **흔한 오해. `pipe` 는 편하지만 손으로 쓴 반복보다 느리다**
>
> 많은 언어에서 반복자 체인은 중간 객체나 간접 호출을 남기거나, 최적화기가 운 좋게 걷어내기를 기대한다. `pipe` 는 한 번 훑기가 정의이고 스테이지 op 은 단형화된 직접 호출이다. `pipe` 하나는 반복 하나로 낮아지고, 비용은 손으로 쓴 반복과 같은 자리에서 생긴다. 편한 쪽과 빠른 쪽이 갈리지 않도록 낱말을 고른 결과다.

24.5 이름이 같은 내장 `map`·`filter`

`pipe` 밖에도 `map`·`filter` 가 있다. 이쪽은 스테이지가 아니라 **문 하나로 슬라이스를 옮겨 담는 내장 연산**이고, 인자가 셋이다 — `map <받는 자리> <op> <원천> .` · `filter <받는 자리> <op> <원천> ..`

```
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
```

`[1,2]` 를 세 칸짜리 `out` 에 담으면 `[2,4,0]` 이다. 거꾸로 `[1,2,3]` 을 두 칸에 담으려 하면 셋째를 쓰는 순간 멈춘다 — 받는 자리가 차도 말없이 멈추는 일은 없다(`collect into` 와 같다). 받는 자리는 `mut slice` 여야 하고(`E-MAP-SINK`), 원소는 스칼라여야 한다(`E-MAP-ELEM`). 낱말은 같지만 `pipe` 안에서는 op 하나만 받는다 — 인자 수를 보면 어느 쪽인지 안다.

24.6 흔한 실수

반례. 스테이지에 식을 적어 람다를 흉내 낸다

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 ✘ 이름 없는 함수를 흉내 냈다 --- 스테이지에는 이름 붙은 op 하나를 건넨다
    filter gt 2 .
    count .
  end .
end
```

실행 결과

```
$ lowentc --check mistake_lambda.low
mistake_lambda.low:8:0 E-FOLD-OP: this `pipe` stage names an op that does not exist
```

다른 언어의 `filter(x => x > 2)` 를 옮기면 `filter gt 2` 가 되기 쉽다. 스테이지는 **이름 붙은 op 하나**를 받으므로 `E-FOLD-OP` 다(진단은 "그런 op 이 없다" 고만 말한다). 조건에 이름을 붙이는 것은 번거로워 보이지만, `over2` 라는 이름이 곧 그 줄의 설명이 되고 같은 조건을 다른 `pipe` 에서도 쓴다.

examples/ch24/lambda_fixed.low

```
module lambda_fixed .
rem run: count_big [1,2,3,4,5]

rem 조건에 이름을 붙인다 --- 이름이 곧 설명이 된다
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
```

실행 결과

```
$ 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]
```

반례. 넓은 값을 내는 `map` 을 좁은 버퍼에 담는다

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 ✘ `u64` 를 내는 `map` 을 `u8` 버퍼에 담는다 --- 1000 이 232 가 된다
    map times1000 .
    collect into out .
  end
  return 0 .
end
```

실행 결과

```
$ 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` 은 `u64` 를 내는데 `out` 은 `u8` 버퍼다. 값을 잃는 좁히기는 적어야만 일어나므로(§ 6.2.5) `E-TYPE-COLLECT` 로 거절된다. 2026-09-16 까지는 통과했고 1000 과 2000 이 조용히 232 와 208 로 담겼다. 좁혀야 한다면 `map` 의 op 안에서 `narrow`·`narrow_wrap`·`narrow_sat` 로 어느 좁히기인지 적어 멈출 자리를 드러낸다.

반례. `fold` 의 op 에서 누산값과 원소의 차례를 바꾼다

examples/ch24/mistake_foldorder.low

```
module mistake_foldorder .
rem expect: E-FOLD-ORDER

rem ✘ 원소를 먼저 받았다 --- 누산값이 `x` 로, 원소가 `acc` 로 들어온다
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
```

실행 결과

```
$ 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` 는 op 에 **누산값을 먼저, 원소를 다음** 에 준다. `add_small_swapped` 는 차례를 거꾸로 받았으므로 `x` 에 누산값이, `acc` 에 원소가 들어온다. 한 걸음이 `누산값 = op(누산값, 원소)` 이므로 첫 입력의 타입은 출력 타입과 같아야 하는데 여기서는 `u8` 과 `u64` 로 다르다 — `E-FOLD-ORDER` 로 거절한다. `fold·scan` 의 op 은 언제나 `input acc … . input x … .` 차례로 적는다.

반례. 멈출 수 있는 스테이지 op 을 쓰고 `fn` 으로 적는다

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 ✘ 스테이지 op 이 멈출 수 있는데 `pipe` 를 쓴 op 을 `fn` 으로 적었다
fn count_checked input xs slice u8 . output u64 .
do
  return pipe xs do
    filter nonzero .
    count .
  end .
end
```

실행 결과

```
$ 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
```

`pipe` 는 스테이지 op 을 부르는 반복이므로, 스테이지 op 의 효과가 `pipe` 를 쓴 op 으로 번진다. `nonzero` 가 `panic` 할 수 있으니 `count_checked` 도 `panic` 을 낸다. 그래서 `fn` 이면 `E-EFFECT-CALC` 다. `proc … effects panic .` 으로 적거나, 멈추는 대신 조건을 만족하지 않는 원소를 걸러 내는 순수한 op 을 쓴다.

흔한 오해. `collect into` 는 버퍼가 차면 남는 것을 버리고 끝난다

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 버퍼는 두 칸 --- 남는 원소가 둘이면 담기고, 넷이면 셋째를 담으려는 순간 멈춘다
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
```

실행 결과

```
$ 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)
```

`[1,3,4]` 에서 남는 원소는 둘(3·4)이라 두 칸에 담긴다. `[3,4,5,6]` 은 넷이라 셋째를 담으려는 순간 **멈춘다**. 예전에는 두 칸에서 말없이 끝났고, 남은 둘이 사라진 것을 아무도 몰랐다. 두 길이를 번역 시점에 알면(`array 5 u8` 입력처럼) 멈출 것도 없이 `E-COLLECT-FULL` 로 거절된다. `pipe` 는 할당하지 않으므로 버퍼를 늘려 주지도 않는다. 들어갈 만큼만 담으려면 `take 2` 를 **적는다**.

24.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`pipe xs do … end`	원천 `xs` 를 한 번 훑는다	뼈대(색인·끝 조건)를 언어가 맡는다
`filter over2 . • map dbl .`	남기기 · 바꾸기 — 이름 붙은 op 을 건넨다	람다가 없다 — 이름이 설명이 된다
`take 2 . • skip 1 .`	앞 몇 개만 · 앞 몇 개 버리기	필요한 만큼만 읽는다
`enumerate idxadd . • zip ys addb .`	순번·짝을 op 의 인자로 건넨다	튜플을 만들지 않는다
`scan 0 addb . • fold 0 addu .`	누적해 흘리기 · 누적해 값 하나	op 은 누산값이 먼저, 원소가 다음

`count .` · `any is_zero .` · `all under10 .`	값을 내는 종결자	`return pipe … end` 로 쓸 수 있다
`collect into out .`	부르는 쪽 버퍼에 담는다	`pipe` 는 할당하지 않는다 — 모자라면 거절되거나 멈춘다
종결자는 정확히 하나, 맨 끝	뒤에 스테이지가 오면 `E-PIPE-NO-TERMINAL`	흐름의 끝이 한곳에 보인다
`sort` **같은 낱말**	없다 — `E-PIPE-STAGE`	융합할 수 없는 연산은 넣지 않았다

표 24.2 — `pipe` 의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`pipe <원천> do … end` 는 스테이지(`filter`·`map`·`take`·`skip`·`enumerate`·`zip`·`scan`)를 거쳐 종결자 (`collect into`·`fold`·`count`·`any`·`all`) 하나로 끝난다. 스테이지에는 이름 붙은 op 을 건네고, 담을 버퍼는 부르는 쪽이 준다. 한 번 훑기와 중간 배열 없음은 정의이며, 융합할 수 없는 연산은 낱말에 없다. `any`·`all`·`take` 는 필요한 만큼만 읽는다.

제7부 — 동시성

25 액터 — 상태를 가진 채 메시지로 사는 것

이 장의 차례

25.1 선언하고, 만들고, 말을 건다 299
25.2 메시지에 값을 싣는다 300
25.3 우편함 — 넣기와 비우기 302
25.4 터지게 두고 다시 세운다 303
25.5 액터를 쓸 수 있는 자리 304
25.6 권한을 든 액터와 상태에 둘 수 있는 것 305
25.7 액터로 설계하기 — 계좌 둘 사이의 이체 307
25.8 흔한 실수 308
25.9 이 장의 문법 한눈에 315

먼저 알아야 할 것

15장 효과 · `state`·`panic` 은 권한 없이 적는 효과다
19장 소유 · 소유를 넘기면 보낸 쪽은 더 쓸 수 없다
20장 할당기와 고정 메모리 · 할당기는 트레이트를 갖춘 액터다

돌아보기

20장의 범프 할당기는 무엇으로 만들고 무엇으로 불렀는가? 할당기의 커서는 어디에 살았는가?

> **답.** `spawn actor allocs.bump_bytes` 로 만들고 `send a reserve 3` 으로 불렀다. 커서는 액터의 상태에 살았고, 바깥에서는 메시지로만 그 상태에 닿았다. 이 장은 그 액터를 제대로 다룬다.

이 장의 필요성과 맥락

제7부는 여러 일이 함께 진행되는 자리를 다룬다. 동시성 결함의 대부분은 **여러 흐름이 같은 상태를 함께 만지는** 데서 나온다. 자물쇠로 막을 수 있지만 자물쇠를 잊거나 차례를 틀리면 조용히 틀린다. 액터는 반대쪽에서 접근한다. 상태를 액터 안에만 두고, 메시지를 한 번에 하나씩 처리하게 한다. 그러면 그 상태를 동시에 만질 방법이 처음부터 없다. 제7부가 액터로 시작하는 것은, 표준 라이브러리의 할당기처럼 이 모양이 이미 이 언어의 곳곳에 쓰이기 때문이다.

이 장이 끝나면

`actor … do state do … end … end` 로 액터를 선언하고, `spawn actor` 로 만들고, `send` 로 메시지를 보내는 법을 익힌다. 액터 안의 op 도 `fn`·`proc` 규칙을 따른다는 것, 메시지에 값을 싣고 소유를 넘기는 법을 알게 된다. 우편함에 넣고 나중에 비우는 `spawn send`·`drain`, 터진 액터를 다시 세우는 `failure restart`, 그리고 액터를 쓸 수 있는 자리를 정하는 `build profile` 도 보게 된다.

이 장에서 답할 질문

1 액터를 부르는 것은 결국 함수 호출과 무엇이 다른가?

25.1 선언하고, 만들고, 말을 건다

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
```

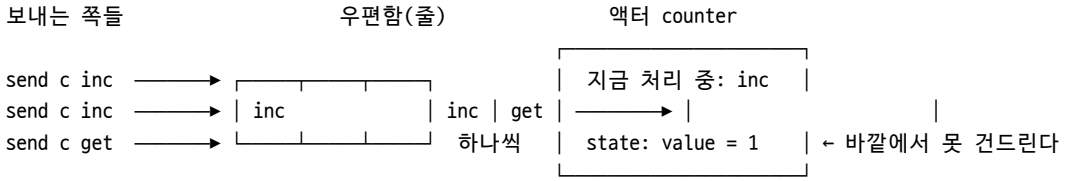
실행 결과

```
$ lowentc --run use_counter counter.low
use_counter() = 2
```

- `actor counter do … end` 가 액터를 선언한다. `state do value u64 . end` 가 액터의 상태다. 상태는 액터 안에만 있고 바깥에서 직접 건드릴 수 없다.
- 상태를 고치는 `inc` 는 `proc` 이고 `effects state` 다. 읽기만 하는 `get` 은 `fn` 이다. 액터 안의 op 도 5장의 갈래 규칙을 그대로 따른다.
- `var c counter be spawn actor counter .` 가 액터 하나를 만든다. 상태는 0 으로 시작한다.
- `send c inc` 는 `c` 에게 `inc` 메시지를 보내고 처리가 끝날 때까지 기다려 결과를 받는다.

액터는 메시지를 **한 번에 하나씩** 처리한다. 그래서 그 안의 상태는 동시에 건드려지지 않는다. 자물쇠를 손으로 걸 필요가 없는 이유다.

창구가 하나뿐인 은행 창구에 빗대면 쉽다. 손님(메시지)은 줄을 서고, 창구 직원(액터)은 한 번에 한 손님만 받는다. 금고(상태)는 창구 뒤에 있어 손님이 직접 열 수 없다.



줄에서 하나씩만 꺼내므로 value 를 두 메시지가 동시에 고치는 일이 생기지 않는다.

상태를 고치면서 fn 이라고 적으면 거절된다.

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
```

실행 결과

```
$ 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
```

진단의 말대로 그 쓰기는 **다음 메시지**에 보인다. 호출자가 결과를 기억해 두거나 차례를 바꾸면 틀린 답이 나온다.

25.2 메시지에 값을 싣는다

메시지 이름 뒤에 값을 나란히 적는다. 받는 쪽에서 액터 자신은 첫 매개변수처럼 다뤄지고, 실은 값이 그 뒤를 잇는다.

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
```

실행 결과

```
$ lowentc --run tally args.low 5 7
tally(5, 7) = 12
```

`send acct deposit a` 는 `deposit` 의 `amount` 에 `a` 를 싣는다. 메시지 op 에도 계약을 적을 수 있다.

소유를 가진 값을 보내면 소유가 넘어간다.

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
```

실행 결과

```
$ 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
```

`j` 를 첫 메시지로 넘긴 순간 `give_twice` 는 `j` 를 더 쓸 수 없다. 두 흐름이 같은 값을 함께 들고 있으면 경합이 생긴다. 소유가 메시지로만 옮겨 다니면 어느 순간에도 그 값을 만질 수 있는 쪽은 하나뿐이고, 경합은 막는 것이 아니라 **있을 자리가 없다.**

> **문. 액터를 부르는 것은 결국 함수 호출과 무엇이 다른가?**
>
> **답.** `send` 는 문법상 호출과 비슷하지만 두 가지가 다르다. 상태가 액터 안에 갇혀 있어 부르는 쪽이 필드를 읽거나 쓸 방법이 없고, 처리가 한 번에 하나씩이라는 것이 보장된다. 액터에게서 빌린 것을 들고 그 액터에게 다시 말을 거는 것도 거절된다 (`E-BORROW-EXCL`). 그 액터가 상태를 고치는 동안 우리가 그 상태를 보고 있게 되기 때문이다.

25.3 우편함 — 넣기와 비우기

`send` 는 처리가 끝날 때까지 기다린다. 결과가 필요 없는 전달은 `spawn send` 로 우편함에 **넣기만** 한다.

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
```

실행 결과

```
$ lowentc --run drive mailbox.low
drive() = 3
$ lowentc --run nodrain mailbox.low
nodrain() = 0
```

`drive` 는 `inc` 를 세 번 넣고 `drain c` 로 우편함을 넣은 차례대로 비운 뒤 3 을 읽는다. `nodrain` 은 비우지 않았으므로 0 이다. 우편함에 든 메시지는 **저절로 처리되지 않는다.**

처리기가 언제 배달할지 스스로 정하지 않는 까닭은 **결정성**이다. 같은 프로그램은 같은 답을 내야 하고, 배달 시점이 곧 답이다. 비우는 자리를 사람이 고르므로 VM 과 네이티브가 같은 차례로 배달하고, 두 백엔드 대조가 그것을 지킨다. 모든 액터의 우편함을 한꺼번에 비우려면 `schedule .` 을 쓴다. `mailbox bounded 2 .` 처럼 우편함 크기를 정하면 넘치는 넣기가 멈추고, `try spawn send` 는 멈추는 대신 오류를 값으로 준다.

> **흔한 오해. 액터는 각자 스레드 하나씩이다**
>
> 액터는 실행 단위의 **모양**이지 스레드가 아니다. 이 판의 처리기에서 `send·spawn send·drain` 은 한 흐름 안에서 결정적으로 배달된다. 액터가 주는 것은 병렬성이 아니라 **상태의 격리**다. 여러 흐름에 일을 나누는 것은 태스크와 채널 (26장), 데이터를 나누어 동시에 계산하는 것은 병렬 되풀이(27장)의 몫이다.

25.4 터지게 두고 다시 세운다

액터의 `op` 이 `panic` 하면 그 액터를 다시 세울 수 있다. 상태가 처음으로 돌아가고 그 메시지가 다시 처리된다.

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
```

실행 결과

```
$ 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` 는 첫 `step` 에서 `bad` 를 1 로 만들고, 둘째 `step` 에서 `panic` 한다. `failure restart max 3 .` 이 있으므로 상태를 처음(`bad = 0`)으로 되돌리고 메시지를 다시 처리해 42 를 낸다.
- `doomed` 는 언제나 터진다. 두 번 다시 세운 뒤에도 터지면 그 실패는 **위로 넘어가** 프로그램이 멈춘다. 조용히 멈추어 있는 액터를 남기지 않는다.

정책은 셋이다 — `restart max <수>`(그 수만큼), `never`(처음 터질 때 바로 위로), `always`(셈하지 않고 계속). 다시 세우는 것은 `panic` 에만 해당한다. 계약 위반은 프로그램이 스스로 적은 약속을 어긴 것이라 다시 해도 같으므로 다시 세우지 않는다. 되살아나는 것은 **자기 상태뿐**이고 다른 액터는 건드리지 않는다.

상태가 어쩌다 이상해진 실패를 하나하나 손으로 되돌리는 코드는 길고, 길면 그 코드 자체가 틀린다. 상태를 처음으로 돌리는 것은 짧고 언제나 맞는 복구다.

25.5 액터를 쓸 수 있는 자리

프로그램은 자기가 어느 자리에서 도는지 `build profile <이름> .` 으로 적을 수 있다. 적으면 그 자리가 감당하지 못하는 동시성을 번역할 때 거절한다.

프로파일	등급	무엇까지
`freestanding`	0	운영체제 없음. 동시성을 쓰지 않는다
`embedded`	1	정해진 일감을 나누어 도는 데까지(흐름이 정적으로 정해진다)
`native`	2	흐름을 만들고 채널로 주고받는다
`server`	3	액터까지

표 25.1 — 프로파일과 여는 등급

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
```

실행 결과

```
$ 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
```

`actor` 를 **선언**하는 것만으로 3 등급이다. 프로파일을 적지 않으면 막지 않는다. 적은 사람만 그 약속을 진다.

25.6 권한을 든 액터와 상태에 둘 수 있는 것

액터의 상태에는 권한 칸을 둘 수 있다. 권한은 번역할 때만 있는 표시라서 그 칸은 실행 중 크기가 0 이다. 대신 그 칸이 무엇으로 채워지는지가 규칙으로 정해진다.

examples/ch25/capfield.low

```
module capfield .
rem run: main

rem 기록 하나마다 고정 창에서 16 바이트를 깎아 두는 장부
actor logbook do
  state do
    rem 권한 칸 — 실행 중 크기가 0 이고, 띄우는 자리의 권한으로 채워진다
    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
```

실행 결과

```
$ lowentc --run main capfield.low
main() = 2
```

- `root cap allocator .` 가 권한 칸이다. `write` 는 그 칸으로 `alloc_bytes root capacity 16` 을 부른다. 권한을 메시지마다 인자로 건네지 않아도 된다.
- `main` 이 `cap allocator` 를 받았기 때문에 `spawn actor logbook` 이 허락된다. 칸은 **띄우는 자리의 권한**으로 채워진다.
- 기록 둘을 남기고 `count` 가 2 를 답한다.

권한을 받지 않은 op 이 같은 액터를 띄우면 거절된다.

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 ✘ cap allocator 를 받지 않은 op 이 권한 칸 액터를 띄운다
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
```

실행 결과

```
$ 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
```

이것이 허락되면 `spawn` 한 줄이 없던 권한을 지어낸다. 권한 칸은 "이 액터를 만든 자리가 이미 그 권한을 쥐었다" 는 사실을 옮겨 적는 것일 뿐이다. 표준 할당기 `allocs.fixed_bytes·heap_bytes` 도 이 규칙을 따르는 평범한 액터다(20장).

상태 칸에 둘 수 있는 것과 없는 것은 이렇다.

타입	받아들이나	까닭
수 · bool · option · 구조체	받는다	값이라서 액터 안에 갇힌다
`slice · mut slice`	받는다	할당기가 받침 바이트를 이렇게 쥔다 (`allocs.bump_bytes`)
`cap allocator · cap heap`	받는다 — 띄우는 op 에 같은 권한이 있어야 한다	`E-CAP-FORGE` 가 지어내기를 막는다
`array <수> <타입>`	거절 — `E-TYPE-ARRAY`	고정 길이 배열은 op 입력에서만 받는다. 길이를 둘 자리가 없다
`ref · mut_ref`	이 판은 받는다 — 그러나 쓰면 멈춘다	무엇을 빌렸는지 적을 자리가 없다. 아래 "흔한 실수" 를 본다

표 25.2 — 액터 상태 칸의 타입

25.7 액터로 설계하기 — 계좌 둘 사이의 이체

지금까지의 조각을 한 설계에 모은다. 계좌마다 액터 하나를 두고, 이체는 두 액터에게 차례로 말을 거는 op 이 맡는다.

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 잔액을 채운다 --- 상태를 고치므로 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 모자라면 상태를 건드리지 않고 오류를 값으로 돌려준다
```

```
  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 두 액터 사이의 이체 --- 빼기가 성공했을 때만 넣는다
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
```

실행 결과

```
$ lowentc --run move transfer.low 30
move(30) = 20030
$ lowentc --run move transfer.low 80
move(80) = 50000
```

- 잔액은 `account` 의 상태에만 있다. 바깥은 `deposit`·`withdraw`·`peek_balance` 세 메시지로만 계좌에 닿는다.
- `withdraw` 는 잔액이 모자라면 **상태를 건드리지 않고** `error insufficient` 를 돌려준다. 멈추지 않는다 — 잔액 부족은 계좌를 쓰는 쪽이 다룰 수 있는 실패다(17장).
- `move` 는 빼기가 성공했을 때만 넣는다. `move 30` 은 보내는 쪽 20, 받는 쪽 30 이라 20030 을 낸다. `move 80` 은 빼기가 실패해 두 계좌가 50 과 0 그대로다(50000).
- 메시지는 한 번에 하나씩 처리되므로 `withdraw` 가 잔액을 확인하고 줄이는 사이에 다른 메시지가 끼어들 수 없다. "확인하고 쓰기" 사이의 경합이 이 모양에는 없다.

`withdraw` 의 `errors insufficient .` 에 조건을 붙이지 않은 것은 일부러다. 아래 "흔한 실수" 의 넷째 항목이 그 까닭이다.

25.8 흔한 실수

> **반례. 크기를 정한 우편함에 비우지 않고 계속 넣는다**
>
> `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 ✘ 우편함은 두 통까지인데 비우기 전에 셋째를 넣는다
  spawn send c inc .
  drain c .
  return send c get .
end
```

실행 결과

```
$ 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 .` 는 “대기 중인 메시지는 둘까지” 라는 약속이다. 셋째 `spawn send` 는 넘치므로 `E-VM-MAILBOX-FULL` 로 멈춘다. 멈추는 대신 다루려면 `try spawn send` 로 넣는다. 그러면 넘침이 `result` 로 돌아오고, 비운 뒤 다시 넣을 수 있다.

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 넣지 못하면 멈추지 않고 오류를 값으로 받는다 --- 비운 뒤 다시 넣는다
  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
```

실행 결과

```
$ lowentc --run burst bounded_fixed.low
burst() = 3
```

반례. 액터가 다루지 않는 메시지를 보낸다

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` 는 `dec` 메시지를 다루지 않는다
  return send c dec .
end
```

실행 결과

```
$ 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)
```

메시지 이름은 **받는 액터의 타입 안에서** 찾는다. `counter` 에는 `dec` 가 없으므로 `E-IR-UNDEF` 다. 진단이 덧붙인 사연대로, 한때는 이름만으로 찾아서 같은 이름의 처리기를 가진 두 액터가 조용히 하나를 나눠 썼다.

반례. 바깥에서 액터의 상태 칸을 읽는다

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 ✘ 액터의 상태 칸을 바깥에서 읽는다 --- 막혀야 할 자리가 이 판에서는 통과한다
  return field c value .
end
```

실행 결과

```
$ 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
```

이 장의 첫 약속은 “상태는 액터 안에만 있고 바깥에서 직접 건드릴 수 없다” 였다. `field c value` 는 그 문을 돌아가려는 시도이고, `E-ACTOR-FIELD` 로 거절된다. 상태가 바깥에서 읽히면 메시지를 한 번에 하나씩 처리한다는 전제가 소용없어진다 — 읽는 쪽이 메시지 사이의 값을 보기 때문이다. 상태가 필요하면 `get` 같은 읽기 메시지를 두고 `send c get` 으로 묻는다.

반례. 메시지 op 의 오류 조건에 상태 칸을 쓴다

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 ✘ 오류 조건이 상태 칸 `balance` 를 읽는다 --- 나갈 때는 이미 줄어든 잔액으로 다시
읽힌다
  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 ✘ 액터 밖에서도 같다 --- 모듈 `var` 와 `mut` 자리의 인자도 몸통이 바꿀 수 있으므로
오류 조건에 서지 못한다
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
```

실행 결과

```
$ 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 .` 는 "잔액보다 많이 빼려 하면 이 오류" 라는 뜻으로 적었다. 그런데 `errors` 의 조건은 op 에 **들어올 때**의 값으로 읽는다(정본 6.4.2). `errors` 는 `requires` 와 같은 쪽 — **부르는 쪽이 무엇을 잘못했는가**를 말하는 절이고, 부르는 쪽이 한 일은 건넨 것뿐이기 때문이다. 그래서 몸통이 바꿀 수 있는 이름(상태 칸 · 모듈 `var` · `mut` 자리의 인자)은 조건에 설 수 없고, 번역이 `E-ERRORS-STATE` 로 거절한다. 나갈 때 값으로 읽으면 성공한 실행이 스스로를 고발한다 — 잔액이 50 에서 20 으로 줄었으므로 나갈 때 `gt 30 20` 이 참이 되어 "조건이 참인데 오류를 내지 않았다" 가 된다. 오류 조건은 **입력**으로 적고, 상태를 보는 판정은 본문의 `guard` 가 맡는다 — 위의 `transfer.low` 처럼 조건 없이 `errors insufficient .` 만 적는다.

반례. 상태 칸에 빌림을 둔다

examples/ch25/mistake_reffield.low

```
module mistake_reffield .
rem expect: E-ACTOR-STATE-REF

actor holder do
  state do
    rem ✘ 빌림을 상태에 둔다 — 무엇을 빌렸는지 적을 자리가 없다
    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
```

실행 결과

```
$ 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`)
```

빌림(ref)은 빌려준 쪽보다 오래 살 수 없다(12장). 액터의 상태는 액터가 사는 동안 남으므로, 그 칸이 무엇을 빌렸는지 적을 자리가 없다. 그래서 선언에서 E-ACTOR-STATE-REF 로 거절한다. 2026-09-16 까지는 선언을 받아들이고 칸을 비운 채 액터를 띄웠고, deref r 에 이르러서야 VM 은 E-VM-TYPE 으로, 네이티브는 panic 으로 멈췄다. 상태에는 빌림 대신 값을 두고, 값이 크면 액터가 사는 동안 유효한 슬라이스를 받아 둔다.

흔한 오해. 다시 세운 액터는 터지기 직전 상태에서 이어 간다

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 값이 2 일 때 한 번 터진다
    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 셋째 메시지에서 터지고, 상태가 0 으로 돌아간 뒤 이 메시지가 다시 처리된다
  return send c inc .
end
```

실행 결과

```
$ lowentc --run three_incs restart_resets.low
three_incs() = 1
```

inc 두 번으로 값이 2 가 된 뒤 셋째 메시지에서 터진다. 다시 세우면 상태는 **처음**(0)으로 돌아가고 그 메시지가 다시 처리되므로 답은 3 이 아니라 1 이다. 터지기 직전의 상태는 바로 그 상태 때문에 터졌을 수 있으므로 믿지 않는다. 잃으면 안 되는 값은 액터 바깥 — 다른 액터나 파일 — 에 두고, 다시 선 액터가 그것을 읽게 한다.

흔한 오해. 같은 액터 타입의 값들은 상태를 나눠 쓴다

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` 는 자기 상태를 따로 가진다 --- 1 이 나온다
  return send right inc .
end
```

실행 결과

```
$ lowentc --run two_counters separate_state.low
two_counters() = 1
```

`spawn actor counter` 를 두 번 하면 상태가 둘 생긴다. `left` 를 두 번 올려도 `right` 는 1 부터 센다. 액터 타입은 설계도이고 상태는 `spawn` 할 때마다 새로 생긴다. 여럿이 같은 값을 봐야 한다면 그 값을 가진 액터 **하나**를 두고 모두가 그 액터에게 말을 건다.

25.9 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`actor counter do state do value u64 . end … end`	상태를 가둔 실행 단위를 선언	상태를 동시에 만질 길이 처음부터 없다
`proc inc … effects state . • fn get …`	상태를 고치는 메시지 · 읽기만 하는 메시지	`fn`·`proc` 규칙이 그대로 — 고치면서 `fn` 이면 `E-EFFECT-PURITY`
`var c counter be spawn actor counter .`	액터 하나를 만든다(상태는 0 에서 시작)	`spawn` 마다 상태가 따로
`send c inc • send acct deposit a`	보내고 처리가 끝날 때까지 기다린다 · 값을 싣는다	액터가 먼저 — 메시지는 액터를 첫 매개변수로 받는 op
`spawn send c inc . • drain c . • schedule .`	우편함에 넣기 · 그 액터의 우편함 비우기 · 모두 비우기	배달 시점을 사람이 고른다 — 결정성

`mailbox bounded 2 .` • `try spawn send`	우편함 크기 · 넘침을 값으로 받기	넘치면 멈추거나 `result`
`failure restart max 3 .` • `never` • `always`	`panic` 한 액터를 처음 상태로 다시 세운다	다 쓰면 실패를 위로 넘긴다
`build profile server .`	액터(3 등급)를 쓸 수 있는 자리	적은 사람만 그 약속을 진다
`state do root cap allocator . … end`	권한 칸 — 실행 중 크기 0	띄우는 op 에 같은 권한이 없으면 `E-CAP-FORGE`

표 25.3 — 액터의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

액터는 상태를 안에 가두고 메시지를 한 번에 하나씩 처리한다. `spawn actor` 로 만들고 `send` 로 기다려 부르며, 메시지 op 은 `fn·proc` 규칙을 따른다. 메시지에는 값을 싣고, 소유 값을 보내면 소유가 넘어간다. `spawn send` 는 우편함에 넣기만 하고 `drain·schedule` 이 비운다. `failure restart` 는 `panic` 한 액터의 상태만 처음으로 돌려 다시 세우고, 다 쓰면 실패를 위로 넘긴다. `build profile` 은 액터를 쓸 수 있는 자리를 정한다.

26 태스크와 채널 — 묶인 흐름끼리 주고받기

이 장의 차례

26.1 흐름은 블록에 묶인다 318
26.2 채널 — 차례 있는 그릇 319
26.3 적힌 것만 봐도 아는 교착 321
26.4 흐름과 메모리 323
26.5 아직 없는 것 325
26.6 흔한 실수 326
26.7 이 장의 문법 한눈에 329

먼저 알아야 할 것

15장 효과 · `concurrent` 는 `wait` 을 딸고 온다
25장 액터 · 상태를 가두고 메시지를 한 번에 하나씩 처리한다
2장 첫 프로그램 · `test` 블록은 `--test` 로 돈다

돌아보기

15장에서 `concurrent` 와 `wait` 은 무엇으로 갈렸는가?

> **답.** `wait` 은 누구의 도움 없이도 언젠가 깨어나는 기다림(커널·장치가 깨운다)이고, `concurrent` 는 **같은 프로그램의 다른 흐름이 진행해야** 끝나는 기다림이다. 그래서 `concurrent` 를 적으면 `wait` 을 함께 적은 것이 된다. 이 장은 그 "다른 흐름" 을 만들고, 흐름끼리 값을 주고받는 법을 다룬다.

이 장의 필요성과 맥락

액터(25장)는 상태를 가두는 모양이었다. 그러나 일을 여러 흐름에 나누고 결과를 모으거나, 생산자와 소비자가 값을 넘겨받는 모양은 액터만으로 적기 번거롭다. 흐름을 다루는 흔한 결함은 셋이다 — 만들어 놓고 잊은 흐름, 짝이 없어 영원히 기다리는 흐름, 서로를 기다리는 교착. Lowent 는 흐름의 수명을 블록에 묶어 첫째를 문법으로 없애고, 둘째와 셋째 가운데 적힌 것만 봐도 알 수 있는 경우를 번역에서 거절한다. 그리고 여러 흐름의 **모든 차례**를 시험하는 도구를 준다.

이 장이 끝나면

`task_group` 안에서 `spawn <op>` 으로 흐름을 만들고 `await` 로 결과를 받는 법을 익힌다. 흐름이 자기를 만든 블록보다 오래 살지 못한다는 규칙을 알게 된다. `channel·chsend·chrecv` 로 값을 주고받고, 채널 연산이 `concurrent` 효과인 까닭을 이해한다. 짝 없는 기다림과 보내는 쪽이 없는 교착이 번역에서 거절되는 모습, 그리고 `schedule explore_interleavings` 시험이 가능한 모든 차례를 돌려 보는 모습을 보게 된다.

이 장에서 답할 질문

1 `task_group cancel_on_error` 는 무엇을 하는가?

26.1 흐름은 블록에 묶인다

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
```

실행 결과

```
$ 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` 는 그 안에서 만든 흐름들을 하나로 묶는다. 블록이 끝나는 자리에서 묶인 흐름이 모두 끝나며, 하나라도 남은 채로 나가지 않는다.
- `spawn square 5` 는 `square 5` 를 도는 흐름을 만들고 그 흐름의 **핸들**을 준다.
- `await h1` 은 그 흐름이 끝날 때까지 기다렸다가 결과를 준다. 기다리는 동안 다른 흐름이 나아간다.

흐름의 수명을 시간 축에 그리면 이렇다.

```
흐름 h1      |   spawn ├── square 5 돈다 ─┤ 끝                       |
흐름 h2      |       spawn ├── square 6 돈다 ──────┤ 끝              |
             |                       await h1 ▲    await h2 ▲         |
             └ 블록이 열린 자리                       블록이 닫힌 자리 ┘
                흐름은 이 두 선 안에서만 산다 --- end 를 지날 때 남은 흐름은 없다
```

흐름은 **자기를 만든 블록보다 오래 살지 못한다.** 그래서 묶는 자리 밖에서 흐름을 만들면 거절된다.

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
```

실행 결과

```
$ 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>`
```

흐름을 만들어 놓고 잊으면 그것은 프로그램이 끝난 뒤에도 남거나, 이미 없어진 것을 만지거나, 아무도 읽지 않는 실패를 낸다. 블록이 수명을 쥐면 그 셋이 문법으로 사라진다 — 잊을 자리가 없기 때문이다.

같은 파일의 `test join_any_order schedule explore_interleavings` 는 두 흐름이 도는 **모든 차례**를 돌려 보고 답이 같은지 확인한다. 결과 줄의 “2 interleavings agree, exhaustive” 가 그것이다.

> **문. `task_group cancel_on_error` 는 무엇을 하는가?**
>
> **답.** 묶인 흐름 하나가 오류로 끝나면, 아직 시작하지 않았거나 기다리는 형제들을 그 오류로 끝난 것으로 처리한다. 다만 취소는 **되감지 않는다**. 이미 일을 낸 흐름의 효과는 되돌아가지 않고, 앞으로 할 일을 하지 않게 할 뿐이다. 어느 형제가 취소되었는지는 차례에 달려 있고, 명세는 그 차례를 정하지 않는다. 오류 자체는 `await` 로 드러난다.

26.2 채널 — 차례 있는 그릇

채널은 흐름 사이에 값을 넣고 빼는 그릇이다. 넣은 차례대로 나오고, 담을 수 있는 수가 정해져 있다.

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
```

실행 결과

```
$ 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` 가 채널을 만든다. `chsend ch n` 은 넣고, `chrecv ch` 는 뺀다.
- 빈 채널에서 빼려는 흐름은 멈추고, 가득 찬 채널에 넣으려는 흐름도 멈춘다. 멈춘 흐름은 상대가 오면 다시 이어 간다.
- 소비자 둘과 생산자 둘을 한 그룹에 묶었다. 어떤 차례로 돌든 `sink` 에는 10 과 32 가 모두 들어가 42 가 된다.

시험 줄의 “58 interleavings agree, exhaustive” 는 네 흐름이 멈추고 이어 가는 차례 58 가지를 **모두** 돌려 보았고 답이 모두 42 였다는 뜻이다. 동시성 결함은 대개 드문 차례에서만 드러나서 몇 번 돌려 보는 시험으로는 잡히지 않는다. 작은 그룹이라면 모든 차례를 도는 편이 확실하다.

채널 연산은 `concurrent` 효과다. 완결이 **다른 흐름의 진행에 달려 있기** 때문이다.

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
```

실행 결과

```
$ 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
```

`effects none` 으로 적은 생산자가 `chsend` 를 부르자 `concurrent·wait` 을 선언하지 않았다고 거절된다. 이 효과가 머리에 있으면, 운영체제가 없어 동료 흐름을 돌려 줄 실행기가 없는 기계에서 이 op 을 쓸 수 없다는 것이 번역할 때 드러난다.

26.3 적힌 것만 봐도 아는 교착

묶는 자리가 흐름을 하나만 만들고 그 흐름이 짝을 기다리면 거절된다.

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
```

실행 결과

```
$ 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)
```

짝이 없는 기다림은 어떤 차례로도 끝나지 않는다. 돌려 보고 매달리기를 기다리지 않고 번역할 때 말한다.

묶인 흐름들이 모두 받기만 하고 아무도 보내지 않아도 거절된다.

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
```

실행 결과

```
$ 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)
```

흔한 오해. 교착 검사가 있으니 이 언어에서는 교착이 일어나지 않는다

이 검사는 **적힌 것만 봐도 아는** 교착만 잡는다. 두 흐름이 서로 다른 채널을 엇갈려 기다리는 교착, 조건에 따라서만 생기는 교착은 번역이 잡지 못한다. 그런 교착은 실행 중에 모든 흐름이 멈추면 처리기가

> E-VM-DEADLOCK 으로 말하고, 작은 경우는 explore_interleavings 시험이 모든 차례를 돌며 찾아낸다. 잡지 못하는 교착이 있다는 사실은 이 규칙이 무엇을 약속하는지 분명히 하는 것이다.

26.4 흐름과 메모리

태스크는 서로 곁에서 돈다. 그래서 18장의 뿌리와 20장의 할당기를 태스크에서 쓸 때는 규칙이 둘 붙는다.

첫째, 뿌리에서 깎는 op 을 태스크로 띄울 수 없다.

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 ✘ 뿌리에서 깎는 op 을 태스크로 띄운다
    spawn worker 1 .
    spawn worker 2 .
  end
  return 0 .
end
```

실행 결과

```
$ 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)
```

뿌리마다 커서는 하나이고, 영역은 **차례대로** 되감긴다. 두 태스크가 영역을 번갈아 열면 한쪽의 end 가 다른 쪽이 아직 쓰는 바이트를 되감는다. 자물쇠로도 고칠 수 없다. 되감기가 기대는 것은 잠금이 아니라 차례이기 때문이다. 그래서 효과에 alloc·heap 이 있는 op 을 태스크로 띄우면 E-ALLOC-TASK 로 거절한다.

둘째, 커서를 원자적으로 옮기지 않는 할당기를 태스크에 건넬 수 없다.

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 ✘ 커서를 원자적으로 옮기지 않는 할당기를 두 태스크에 건넨다
    spawn user b .
    spawn user b .
  end
  return 0 .
end
```

실행 결과

```
$ lowentc --check mistake_tasksshared.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)
```

두 태스크가 같은 `b` 에서 `reserve` 하면 커서 하나를 두고 경합한다. 할당기의 `reserve` 가 `atomic` 을 적지 않았으면 `E-ALLOC-SHARED` 다. 무는 것은 **나눠 쓰는 것**이다 — 같은 할당기를 태스크 둘에 건네거나, 태스크에 건넨 채 곁에서 또 쓰는 것. 따로 띄운 할당기를 태스크 마다 하나씩 주는 것은 통한다. 또 하나의 길은 **바이트**를 넘기고 할당기는 태스크 안에서 만드는 것이며, 아래가 그것이다.

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 태스크가 받는 것은 할당기가 아니라 바이트다. 할당기는 태스크 안에서 만든다
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 버퍼를 겹치지 않는 두 조각으로 나눠 태스크마다 하나씩 준다
  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
```

실행 결과

```
$ lowentc --run share taskshared_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` 는 버퍼를 겹치지 않는 두 조각으로 나눠 태스크마다 하나씩 준다. 각 태스크는 자기 조각 위에 할당기를 띄워 4 바이트를 쓰고, 두 답을 더하면 8 이다. 조각이 겹치지 않으니 경합할 커서도, 되감길 바이트도 없다.

26.5 아직 없는 것

끝이 없는 채널(`channel … unbounded`)과 여러 흐름이 나누어 가지는 자물쇠 상태(`lock·rwlock`)는 아직 받아들이지 않는다 (`E-CHAN-UNBOUNDED·E-LOCK-NOTYET`). 없는 것을 받아들이고 나중에 짓는 길도 있지만, 그러면 그 사이에 쓴 프로그램이 되는 줄 알았다가 안 되는 것을 겪는다. 없으면 없다고 말하는 편이 정직하다. 여러 흐름이 같은 메모리를 원자적으로 다루는 길은 27장이 다룬다.

examples/ch26/lock.low

```
module lock .
rem expect: E-LOCK-NOTYET

rem ✘ 흐름끼리 나누는 자물쇠 상태 — 이름은 정본에 있지만 아직 받지 않는다
fn read_count input c lock u64 . output u64 . do
  return 0 .
end
```

실행 결과

```
$ 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` 은 정본에 이름이 있는 타입이라 "없는 타입" 이라고 말하지 않는다. 대신 `E-LOCK-NOTYET` 으로 "아직 짓지 않았다" 고 말한다. 그 사이에 나누어 쓸 수가 필요하면 원자 연산(27장)이나 액터(25장)를 쓴다.

26.6 흔한 실수

반례. 두 흐름이 서로에게서 받기부터 한다

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 ✘ 둘 다 받기부터 한다 --- 첫 값을 넣어 줄 흐름이 없다
    spawn relay a b .
    spawn relay b a .
  end
  return 0 .
end
```

실행 결과

```
$ 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` 는 `a` 에서 받아 `b` 로 넘기고, `relay b a` 는 그 반대다. 둘 다 받기부터 하므로 누구도 첫 값을 넣지 못한다. 흐름이 둘이고 채널이 엇갈려서, 적힌 것만 봐서는 교착인지 알 수 없으므로 번역은 통과한다. 실행하면 모든 흐름이 멈춘 것을 처리기가 보고 `E-VM-DEADLOCK` 으로 알린다. 엇갈린 기다림을 만들지 않으려면 흐름마다 "누가 먼저 보내는가" 를 정해 두고, 가능하면 값이 한 방향으로만 흐르게 짠다.

반례. 두 흐름에 같은 변수를 쓰기로 빌려준다

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 ✘ 두 흐름에 같은 변수를 쓰기로 빌려준다
    spawn bump (mut_ref n) .
    spawn bump (mut_ref n) .
  end
  return n .
end
```

실행 결과

```
$ lowentc --check mistake_sharedvar.low
14:0 E-EXCL: exclusivity violation: overlapping borrow/owner access (readers-XOR-writer)
```

두 흐름이 같은 `n` 을 동시에 올리면 데이터 경합이다. 쓰기 빌림은 하나여야 한다는 규칙(12장)이 흐름 사이에서도 그대로라서 `E-EXCL` 로 거절된다. 첫 줄의 `W-EFFECT-OVER` 는 이 장의 주제와 상관없는 이 판의 결함이다(`mut_ref` 로 쓰는 것을 `state` 로 세지 않는다). 흐름은 자기 몫을 **돌려주고**, 합치는 일은 묶는 쪽이 `await` 뒤에 한다.

examples/ch26/sharedvar_fixed.low

```
module sharedvar_fixed .
rem run: count2

rem 흐름은 자기 몫을 돌려주고, 합치는 일은 묶는 쪽이 `await` 뒤에 한다
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
```

실행 결과

```
$ lowentc --run count2 sharedvar_fixed.low
count2() = 2
```

흔한 오해. 한 번 돌려서 맞으면 동시성 코드도 맞다

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 처음 받은 값만 남긴다 --- 답이 도착 차례에 달린다
    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
```

실행 결과

```
$ 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` 은 처음 받은 값만 남긴다. `drive` 를 돌리면 32 가 나오고, 다른 차례라면 10 이 나온다. 한 번 돌려 본 결과는 그 한 차례의 답일 뿐이다. `schedule explore_interleavings` 시험은 여덟 가지 차례를 모두 돌려 보고, 답이 차례에 따라 갈린다는 것을 `E-SCHED-NONDET` 으로 알린다. 결정적인 프로그램은 모든 차례에서 같은 답을 내야 한다. 받은 값을 모두 더하는 `chan.low` 의 `sink` 처럼, 차례가 달라도 결과가 같도록 처리기를 짠다.

26.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`task_group do … end`	흐름들을 블록에 묶는다 — 끝에서 모두 끝난다	만들어 놓고 잊은 흐름이 문법으로 사라진다
`var h1 u64 be spawn square 5 .`	흐름을 만들고 핸들을 받는다	묶는 자리 밖이면 `E-SPAWN-SCOPE`
`await h1`	그 흐름의 결과를 기다려 받는다	기다리는 동안 다른 흐름이 나아간다
`task_group cancel_on_error do … end`	오류 하나에 형제를 취소한다	취소는 되감지 않는다
`var ch u64 be channel u64 .`	크기가 정해진 차례 있는 그릇	끝없는 채널은 아직 없다 — `E-CHAN-UNBOUNDED`
`chsend ch n` · `chrecv ch`	넣기 · 빼기 — 차거나 비면 멈춘다	`concurrent` 효과 — 완결이 남에게 달렸다
짝 없는 기다림 · 보내는 쪽 없는 받기	번역에서 거절(`E-CONC-ALONE` · `E-CONC-DEADLOCK`)	적힌 것만 봐도 아는 교착
`test … schedule explore_interleavings . do … end`	가능한 모든 차례를 돌려 답을 맞댄다	드문 차례의 결함을 시험이 찾는다
태스크 op 의 효과에 `alloc` · `heap`	`E-ALLOC-TASK`	뿌리의 되감기는 차례에 기댄다 — 잠금으로 못 지킨다

원자적이지 않은 할당기를 `spawn` 인자로	`E-ALLOC-SHARED`	바이트 조각을 넘기고 할당기는 태스크 안에서 만든다
`lock t` · `rwlock t` · `shared_read t`	`E-LOCK-NOTYET`	이름은 정본에 있다 — 짓지 않은 것을 받아 주지 않는다

표 26.1 — 태스크와 채널의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`task_group` 안에서 `spawn <op>` 은 흐름을 만들고 핸들을 주며, `await` 가 결과를 받는다. 흐름은 자기를 만든 블록보다 오래 살지 못하므로 묶는 자리 밖의 `spawn` 은 거절된다. `channel·chsend·chrecv` 는 크기가 정해진 차례 있는 그릇이고 채널 연산은 `concurrent` 효과다. 짝 없는 기다림과 보내는 쪽이 없는 교착은 번역에서 거절되고, `schedule explore_interleavings` 시험은 모든 차례를 돌려 답을 맞댄다.

27 병렬 되풀이와 원자 연산

이 장의 차례

27.1 나눌 수 있다고 밝힌다 332
27.2 처리기가 확인하는 세 조건 333
27.3 모으는 연산은 결합적이어야 한다 334
27.4 원자 연산 335
27.5 기억 차례 336
27.6 레인 — 값 여럿을 한 번에 셈한다 337
27.7 자리를 어떻게 쓰는지 적는다 — access 341
27.8 흔한 실수 341
27.9 이 장의 문법 한눈에 345

먼저 알아야 할 것

12장 빌리기 · 읽기 여럿 또는 쓰기 하나
26장 태스크와 채널 · 흐름은 블록에 묶이고 모든 차례를 시험할 수 있다
16장 권한 · `effects atomic` 은 `cap atomic` 과 짝이다

돌아보기

12장의 배타 규칙(“읽기 여럿 또는 쓰기 하나”)이 멀티스레드 프로그램과 무슨 관계가 있다고 했는가?

> **답.** 한 흐름 안에서 그 규칙이 지켜지면, 여러 흐름으로 나누었을 때 데이터 경합이 없다는 것까지 따라 나온다고 했다. 이 장은 그 성질을 실제로 쓴다 — 되풀이 하나를 조각으로 나누어 여러 흐름이 함께 돌게 하고, 나누어도 답이 같다는 것을 번역이 확인한다.

이 장의 필요성과 맥락

같은 계산을 여러 코어에 나누는 것은 가장 흔한 병렬화이고, 가장 흔한 결함의 자리다. 나눈 조각이 서로의 원소를 읽거나, 같은 변수에 누적하거나, 차례에 따라 답이 달라지는 연산을 쓰면 “빠르지만 가끔 틀린” 프로그램이 된다. Lowent 는 나눌 수 있다는 선언을 **믿지 않고 확인한다**. 확인할 수 없으면 나누지 않는 것이 아니라 번역을 거절한다. 그리고 여러 흐름이 정말로 같은 메모리를 만져야 하는 드문 자리에는 원자 연산을 이름으로 고르게 한다. 제7부의 마지막 장이다.

이 장이 끝나면

op 머리의 `parallel <슬라이스> split .` 으로 되풀이를 나눌 수 있다고 밝히는 법과, 처리기가 확인하는 세 조건(자기 몫만 읽기·자기 몫만 쓰기·걸음을 넘어 사는 자리에 쓰지 않기)을 익힌다. 누적을 `reduce <자리> <연산> .` 으로 밝히고, 결합적이지 않은 연산이 거절되는 이유를 알게 된다. `atomic_add`·`atomic_load` 같은 원자 연산과 기억 차례(`order`), 그 조합 규칙도 보게 된다.

이 장에서 답할 질문

1 원자 연산이 있는데 자물쇠(`lock`)는 왜 없는가?

27.1 나눌 수 있다고 밝힌다

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
```

실행 결과

```
$ 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` 의 머리에 `parallel s split .` 이 있다. "이 되풀이는 `s` 를 조각으로 나누어 여럿이 함께 돌아도 된다" 는 선언이다. 각 걸음은 `index s i` — 자기 원소 — 만 읽고 쓴다.
- `total` 은 합을 누적한다. 누적 변수 `acc` 는 걸음을 넘어 살기 때문에 그대로는 나눌 수 없다. `reduce acc add .` 로 "조각마다 따로 누적한 뒤 `add` 로 합친다" 고 밝힌다.

두 선언이 무엇을 허락하는지 그림으로 보면 이렇다.

```
split --- 조각마다 제 몫만 만진다
  s:  [ 0 1 2 3 ¦ 4 5 6 7 ¦ 8 9 10 11 ]
        조각 A     조각 B     조각 C        ← 셋이 동시에 돌아도 서로 닿지 않는다

reduce acc add --- 조각마다 따로 모은 뒤 합친다
  조각 A: acc_A = 0+1+2+3      = 6  ┐
  조각 B: acc_B = 4+5+6+7      = 22 ┼ add → acc = 66
  조각 C: acc_C = 8+9+10+11    = 38 ┘
```

`add` 는 묶는 차례가 달라도 답이 같으므로(결합적) 어떻게 나누어도 순차 결과와 같다. 부동소수 덧셈은 그렇지 않아서 `reduce` 로 나눌 수 없다(아래 «모으는 연산은 결합적이어야 한다»).

처리기가 이 선언을 확인하면 `W-PAR-OK` 로 알린다. 그 알림에는 중요한 말이 들어 있다. VM 은 여전히 순차로 돈다. 그것이 **올바른 구현**인 것은, 나눌 수 있는 조건을 만족하면 병렬 결과가 순차 결과와 비트까지 같다는 정리가 증명되어 있기 때문이다(45장). 네이티브 코드는 실제로 조각을 나누어 여러 스레드로 돈다. 두 백엔드가 같은 답을 내는 것이 곧 그 정리의 실측이다.

27.2 처리기가 확인하는 세 조건

조건	어기면	무엇이 어긋나는가
자기 몫만 읽는다	`E-PAR-READ`	남의 자리를 읽으면 그 값이 옛것인지 새것인지가 누가 먼저 도느냐에 달린다
자기 몫만 쓴다	`E-PAR-WRITE`	두 걸음이 같은 자리에 쓰면 남는 값이 차례에 달린다
걸음을 넘어 사는 자리에 쓰지 않는다	`E-PAR-CARRY`	그런 자리는 걸음들을 묶는다. 모아야 하면 `reduce` 로 밝힌다

표 27.1 — 나누어 돌 수 있는 되풀이의 조건

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
```

실행 결과

```
$ 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
 = ∅)
```

모든 원소에서 첫 원소를 빼는 되풀이다. 첫 조각이 `index s 0` 을 먼저 바꾸면, 다른 조각이 읽는 첫 원소는 이미 바뀐 값이다. 순차로 돌려도 첫 걸음 이후에는 0 을 빼게 되는 결함이 있는데, 나누면 그 결함이 차례에 따라 달라진다.

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
```

실행 결과

```
$ 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> .`
```

`reduce` 없이 `acc` 에 누적했다. 진단이 고치는 법을 그대로 알려 준다.

> **흔한 오해. 병렬로 돌려서 답이 가끔 다르면 그건 성능 조정의 문제다**
>
> 답이 차례에 따라 달라지는 프로그램은 느린 프로그램이 아니라 틀린 프로그램이다. Lowent 에서 `parallel` 절은 뜻을 바꾸지 않는다. 나누어 돈 답과 하나씩 돈 답은 언제나 같아야 하고, 그 조건을 확인할 수 없으면 번역이 거절한다. “빠르지만 가끔 틀림” 은 이 언어가 파는 물건이 아니다.

27.3 모으는 연산은 결합적이어야 한다

`reduce` 로 모을 때, 조각을 어떻게 나누느냐에 따라 합치는 나무의 모양이 달라진다. 연산이 결합적이면 모양이 답을 바꾸지 않고, 결합적이지 않으면 바꾼다.

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
```

실행 결과

```
$ 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` 는 결합적이지 않다. `(1000 - 1) - 2` 와 `1000 - (1 - 2)` 는 다르다. 두 명제 — 결합적이면 모양이 상관없고, 결합적이지 않으면 모양이 결과를 바꾼다 — 는 둘 다 Coq 로 증명되어 있고, 진단이 그 정리의 이름을 인용한다. 부동소수의 덧셈도 차례에 따라 답이 달라지므로 따로 막는다(`E-PAR-FLOAT`).

27.4 원자 연산

나누어 돈 조각들이 **한 자리**를 함께 갱신해야 할 때가 있다. 조각마다 센 개수를 공유 카운터에 더하는 경우다. 보통의 `add` 로는 두 스레드가 같은 값을 읽고 각자 더해 써서 하나가 사라진다. 원자 연산은 쪼개지지 않는다 — 다른 흐름이 그 중간을 볼 수 없다.

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
```

실행 결과

```
$ lowentc --run main counter.low
main() = 10
```

- `atomic_add counter 0 1` 은 슬라이스 `counter` 의 0 번 자리에 원자적으로 1 을 더한다. 자리는 **슬라이스와 색인**으로 가리킨다.
- `count_par` 는 `data` 를 나누어 돌면서 공유 카운터를 갱신한다. 열 바이트를 세었으므로 10 이다.
- 원자 연산은 `atomic` 효과이고, `cap atomic` 을 받아야 한다. `main` 은 시작점에서 `cap atomic` 을 받는다.
- `view_array u64` 는 할당받은 8 바이트를 베끼지 않고 `u64` 하나의 슬라이스로 본다(13장).

권한 없이 원자 효과를 적으면 거절된다.

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
```

실행 결과

```
$ 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))
```

원자 연산은 공짜가 아니다. 한 흐름만 만지는 값에 쓰면 느려지기만 한다. 그래서 기본으로 두지 않고 이름으로 고르게 하며, 그 비용이 머리의 효과와 권한에 적힌다.

> **문. 원자 연산이 있는데 자물쇠(lock)는 왜 없는가?**
>
> **답.** 나누어 가지는 자물쇠 상태는 이 판에서 아직 짓지 않았고, 쓰면 `E-LOCK-NOTYET` 으로 없다고 말한다. 대신 표준 라이브러리에 원자 연산으로 지은 자료구조가 있다. `spsc` 는 락 없이 한 생산자와 한 소비자가 값을 넘기는 링 버퍼이고, 그 정확성은 약한 메모리 모델의 증명을 빌려 확인되었다(34·47장).

27.5 기억 차례

원자 연산 뒤에 `order <이름>` 을 붙여 다른 흐름에게 무엇이 언제 보이는지 정할 수 있다.

이름	뜻
`seq_cst`	모든 흐름이 같은 하나의 차례를 본다. 가장 강하고, 적지 않으면 이것이다
`acq_rel`	읽고 쓰는 연산에서 앞뒤 모두 새어 나가지 않는다
`acquire`	이 읽기 뒤의 일이 앞으로 새어 나가지 않는다
`release`	이 쓰기 앞의 일이 뒤로 새어 나가지 않는다
`relaxed`	원자성만 있고 차례는 없다

표 27.2 — 기억 차례

가장 강한 것이 기본인 까닭은 그것이 가장 추론하기 쉽기 때문이다. 모든 접근이 `seq_cst` 이면 순서대로 사고해도 된다는 것이 증명되어 있다. 약하게 하는 것은 그 값을 아는 사람이 적어서 고르는 일이다.

연산에 따라 뜻이 없는 조합은 거절된다.

차례 넷을 짝이 맞는 연산에 쓰면 이렇다.

examples/ch27/orders.low

```
module orders .
rem run: main
```

```
rem 적지 않으면 seq_cst — 모든 흐름이 같은 차례를 본다
rem 읽고 쓰는 연산(atomic_swap)에는 acq_rel, 쓰기에는 release, 읽기에는 acquire
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
```

실행 결과

```
$ lowentc --run main orders.low
main() = 15
```

`atomic_store … order seq_cst` 로 5 를 쓰고, 읽고 쓰는 `atomic_swap … order acq_rel` 로 7 과 바꾸며 옛값 5 를 얻는다. `order release` 로 15 를 쓰고 `order acquire` 로 읽어 15 를 돌려준다. 짝이 맞지 않는 조합은 거절된다.

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
```

실행 결과

```
$ 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
```

읽기가 무엇을 내보낸다는 말인가. C 는 이런 조합을 정의되지 않은 동작으로 둔다. 쓰기는 `acquire` 일 수 없고, 차례만 정하는 울타리(`atomic_fence`)는 `relaxed` 일 수 없다.

27.6 레인 — 값 여럿을 한 번에 셈한다

`parallel` 이 되풀이를 **흐름** 여럿에 나눈다면, `vec` 은 한 흐름 안에서 **값 여럿**을 한 값으로 들고 한 번에 셈한다(SIMD). `vec u32 4` 는 `u32` 네 개를 담는 레인 넷짜리 값이고, 레인 수는 타입의 일부다. 기계가 한 번에 셈하든 하나씩 셈하든 답은 같다(정본 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 네 레인을 한 값으로 읽고, 5 보다 큰 레인은 5 로 눌러 모두 더한다
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 레인을 가로질러 가장 큰 것 · 가장 작은 것 · 모두 곱한 것
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 레인의 차례를 뒤집고, 한 칸 돌려 메모리에 쓴다
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 이 기계가 u32 레인을 한 번에 몇 개 다루나 --- 번역 시점의 수
fn lanes_here output u64 .
do
  return native_lanes u32 .
end
```

실행 결과

```
$ 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` 은 슬라이스의 0 번 자리부터 레인 넷을 읽는다. `store ys 0 t` 는 거꾸로 쓴다.
- `splat 5` 는 모든 레인에 5 를 채운다. 레인 수는 담는 이름의 타입(`vec u32 4`)이 알려 준다.
- `gt v lim` 처럼 `vec` 끼리 비교하면 레인마다 참거짓이 담긴 **가림막** `mask 4` 가 나온다. `select over lim v` 는 가림막이 켜진 레인에서 `lim` 을, 꺼진 레인에서 `v` 를 고른다. 갈래(`if`) 없이 레인마다 고르므로 기계가 한 명령으로 처리한다.
- `reduce_add·reduce_max·reduce_min·reduce_mul` 은 레인을 가로질러 하나로 모은다. `[1,9,3,7]` 을 5 로 누르면 `[1,5,3,5]` 라서 합이 14 다.
- `reverse` 는 레인의 차례를 뒤집고, `rotate r 1` 은 한 칸 돌린다. `[7,3,9,1]` 을 돌린 `[3,9,1,7]` 이 메모리에 쓰였다.
- `native_lanes u32` 는 이 기계가 `u32` 레인을 한 번에 몇 개 다루는지를 **번역 시점에** 준다. 레인 수를 그 수로 고르면 한 번에 더 많이 셈할 뿐 답은 같다.

  가림막으로 일부 레인만 읽거나 쓸 수도 있다.

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 5 보다 큰 레인만 쓴다 --- 가려진 레인의 자리는 건드리지 않는다
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 켜진 레인만 읽고 나머지 레인에는 기본값 100 을 둔다
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
```

실행 결과

```
$ 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` 은 켜진 레인(9 와 7)의 자리에만 쓰고 나머지 자리는 건드리지 않는다. `load_masked xs 0 m fallback` 은 켜진 레인만 읽고 꺼진 레인에는 기본값 100 을 둔다. 슬라이스 끝에 네 칸이 다 남지 않을 때 꼬리를 처리하는 모양이 이것이다.

op	무엇을 하나	적어 둘 것
`load` · `store` · `load_masked` · `store_masked`	메모리와 레인 사이를 읽고 쓴다	가림막 판은 켜진 레인만
`splat` · `select`	한 값을 모든 레인에 · 가림막으로 레인마다 고르기	`splat` 은 타입 문맥이 필요하다(아래)
`reduce_add` · `reduce_max` · `reduce_min` · `reduce_mul`	레인을 하나로 모은다	결과는 원소 타입
`reverse` · `rotate`	레인의 차례를 뒤집는다 · 돌린다	칸 수는 번역 시점 상수
`native_lanes`	이 기계의 레인 수(번역 시점)	답을 바꾸지 않는다
`sum_neumaier` · `sum_seq`	`view_array` 로 본 부동소수 조각을 모두 더한다	레인 op 이 아니다 — 이름이 더하는 방법을 말한다(보정 · 앞에서 뒤로)
`avg`	레인마다 반올림하는 평균	값이 `(a+b+1)>>1` 로 정해져 있다 — 레인이 넘치지 않도록 넓혀 더한다
`prefetch xs i`	곧 쓸 자리를 미리 캐시로 끌어 온다	뜻을 바꾸지 않는 성능 힌트

표 27.3 — 레인과 배열을 다루는 내장 op

두 가지를 조심한다. 첫째, `splat` 은 레인 수를 **선언된 타입에서** 받으므로 식 안에 바로 쓸 수 없다. 둘째, **레인을 더하는 것과 조각을 더하는 것은 다른 op 이다** — 레인은 `reduce_add`, 부동소수 조각은 `sum_neumaier`·`sum_seq` 다. 2026-09-17 까지는 뒤엣것의 이름이 `sum`·`sum_fast` 였고, 정본은 같은 이름으로 «레인을 더한다» 를 적고 있었다. 한 이름이 두 가지를 뜻했으므로 이름을 갈랐다(정본 6.3.7.1).

반례. `splat` 을 식 안에 바로 적는다

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` 을 식 안에 바로 적었다 --- 레인 수를 말해 줄 타입이 없다
  var over mask 4 be gt v (splat 5) .
  return reduce_add (select over v v) .
end
```

실행 결과

```
$ 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` 은 한 값을 모든 레인에 채우는데, **레인이 몇인지**는 선언된 타입만이 말한다. 식 안에는 그것을 말해 줄 자리가 없어 값이 스칼라로 읽히고, 감싼 비교가 `mask` 타입과 어긋난다. `E-VEC-SPLAT` 으로 거절하며, 고치는 길은 레인 수를 적은 이름에 먼저 담는 것이다 — `var lim vec u32 4 be splat 5 .` 뒤에 `gt v lim` 이다.

27.7 자리를 어떻게 쓰는지 적는다 — access

`access <이름> <모드> .` 절은 op 이 입력 자리를 **읽기만** 하는지, **쓰기만** 하는지 머리에 적는다. 부르는 쪽과 스케줄러가 그 약속을 믿고 판단한다.

examples/ch27/access.low

```
module access .
rem run: peek [9,8,7]

rem access data shared_read — 이 op 은 data 를 읽기만 한다. 여러 태스크가 함께 쥘 수 있
다는 약속이다
fn peek input data slice u8 . output u64 .
  access data shared_read .
do
  return widen u64 (index data 0) .
end
```

실행 결과

```
$ lowentc --run peek access.low [9,8,7]
peek([9,8,7]) = 9
  arg0 (written) = [9,8,7]
```

`shared_read` 는 “읽기만 한다” 이다. 쓰기가 없으면 경합도 없으므로 여러 태스크가 같은 자리를 함께 쥘 수 있다. `write_only` 는 “쓰기만 한다” 이고, 아직 채워지지 않은 버퍼를 넘겨도 된다는 뜻이다. 두 모드는 도구가 몸을 보고 **검사한다**. `sequential` 같은 나머지 모드는 커널 스케줄링 힌트라 아직 아무것도 강제하지 않으며, 적으면 `W-NOT-YET` 이 그렇다고 말한다.

27.8 흔한 실수

반례. 나누어 도는 조각들이 공유 카운터를 보통 연산으로 올린다

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 ✘ 모든 조각이 같은 자리 `index counter 0` 에 읽고 더해 쓴다
    set (index counter 0) (wrap_add (index counter 0) 1) .
    set i (add i 1) .
  end
  return .
end
```

실행 결과

```
$ 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 =
 ∅)
```

조각마다 `index counter 0` 을 읽고 1 을 더해 쓴다. 두 스레드가 같은 값을 읽고 각자 더해 쓰면 하나가 사라진다. 이것은 자기 몫이 아닌 자리에 쓰는 것이라 `E-PAR-WRITE` 로 거절된다. 공유 자리를 함께 갱신해야 하면 `cap atomic` 을 받고 `atomic_add counter 0 1` 을 쓴다(이 장의 `counter.low`). 대개는 그보다 조각마다 센 값을 `reduce` 로 모으는 편이 빠르다.

반례. 부동소수의 합을 나누어 모은다

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 ✘ 부동소수의 덧셈은 묶는 차례에 따라 답이 달라진다
  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
```

실행 결과

```
$ 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
```

부동소수의 덧셈은 결합적이지 않다. 조각을 어떻게 나누느냐에 따라 반올림이 달라져 답이 코어 수에 달린다. 그래서 `E-PAR-FLOAT` 다. 진단이 권하는 대로 순차의 `sum_neumaier`(오차를 보정하는 합)를 쓴다. 결정성은 성능 옵션이 아니라 뜻의 일부다.

반례. `reduce` 의 누산을 항등원이 아닌 값에서 시작한다

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 ✘ 누산을 100 에서 시작한다 --- 나누면 조각마다 100 에서 시작한다
  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
```

실행 결과

```
$ 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
```

순차로 돌면 `total [1,2,3,4,5,6]` 은 100 + 21 = 121 이다. 나누어 돌면 **조각마다** `acc` 가 100 에서 시작한다. 네이티브에서 여섯 조각으로 나누면 621 이 나왔다 — 나눈 답이 순차와 같아야 한다는 약속이 깨지는 자리다. 그래서 `E-PAR-IDENTITY` 로 거절한다(2026-09-16 까지는 `W-PAR-OK` 로 통과했다). `reduce` 의 시

작값은 모으는 연산의 **항등원**이어야 한다 — `add`·`bit_or`·`bit_xor` 는 0, `mul` 은 1, 부호 없는 폭의 `max` 는 0. 더할 값이 있으면 되풀이 밖에서 결과에 더한다.

반례. 나눌 되풀이를 다른 모양으로 적는다

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 ✘ 나눌 되풀이는 `while lt i (len s)` 모양이어야 한다 --- 이웃 원소를 읽는 되풀이이
기도 하다
  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
```

실행 결과

```
$ 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
```

처리기는 `while lt i (len s) . do … end` 모양의 되풀이만 나눌 대상으로 알아본다. `while lt (add i 1) (len s)` 는 그 모양이 아니라서 `E-PAR-NOLOOP` 다. 진단의 말대로 `parallel` 절은 **주장**이고, 나눌 되풀이를 찾지 못하면 아무것도 확인하지 못한 채 주장만 남는다. 게다가 이 되풀이는 이웃 원소 `index s (add i 1)` 을 읽는다. 모양을 고쳐도 `E-PAR-READ` 로 거절될 것이다 — 이웃을 읽는 계산(평활화 따위)은 결과를 다른 슬라이스에 쓰는 순차 되풀이로 적는다.

반례. `write_only` 라고 적고 읽는다

examples/ch27/mistake_access.low

```
module mistake_access .
rem expect: E-ACCESS-MODE

rem ✘ write_only 라고 적고 읽는다 — 아직 채워지지 않은 자리일 수 있다
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
```

실행 결과

```
$ 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)
```

`write_only` 자리는 채워지지 않았을 수 있으므로, 읽으면 쓰레기를 읽는 것이다. 부르는 쪽은 그 선언을 믿고 빈 버퍼를 넘긴다. 그래서 `E-ACCESS-MODE` 로 거절한다. 함께 붙은 `W-EFFECT-OVER` 는 호출자 버퍼 쓰기를 효과로 세는 판정이 갈라진 결함 탓이다(24장). 읽어야 한다면 모드를 지우고 보통 입력으로 받는다.

흔한 오해. `reduce` 로 모을 수 있는 것은 덧셈뿐이다

examples/ch27/max_gather.low

```
module max_gather .
rem run: biggest [3,9,2,7]
rem `max` 도 결합적이고 0 이 항등원이라 나누어 모을 수 있다
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
```

실행 결과

```
$ lowentc --run biggest max_gather.low [3,9,2,7]
biggest([3,9,2,7]) = 9
  arg0 (written) = [3,9,2,7]
```

결합적인 연산이면 모을 수 있다. `max` 는 결합적이고 `u64` 에서 0 이 항등원이라 조각마다 가장 큰 값을 구해 다시 `max` 로 합쳐도 답이 같다. `min`·`mul`·비트 연산도 같은 원리다. 거절되는 것은 `sub` 처럼 묶는 차례가 답을 바꾸는 연산(`E-PAR-ASSOC`)과 부동소수 연산(`E-PAR-FLOAT`)이다.

27.9 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`parallel s split .`**(op 머리)**	`s` 를 조각으로 나누어 여럿이 돌아도 된다는 선언	주장을 믿지 않고 확인한다 — 확인되면 `W-PAR-OK`
`while lt i (len s) . do … end`	나눌 수 있는 되풀이의 모양	다른 모양이면 `E-PAR-NOLOOP`

index s i 만 읽고 쓰기	자기 몫만	남의 자리는 E-PAR-READ · E-PAR-WRITE
reduce acc add .	조각마다 누적한 뒤 연산으로 합친다	시작값은 항등원 — 연산은 결합적(E-PAR-ASSOC)
atomic_add counter 0 1 • atomic_load cells 0	슬라이스와 색인으로 가리킨 자리를 원자적으로	effects atomic + cap atomic
… order seq_cst • acq_rel • acquire • release • relaxed	기억 차례 — 적지 않으면 seq_cst	가장 추론하기 쉬운 것이 기본
읽기에 order release 따위	거절(E-ATOMIC-ORDER)	뜻 없는 조합을 정의되지 않은 동작으로 두지 않는다
view_array u64 bytes	바이트를 베끼지 않고 u64 슬라이스로 본다	원자 칸을 할당받은 창 위에 둔다
var v vec u32 4 be load xs 0 . • reduce_add v	레인 넷을 한 값으로 읽기 · 레인 모으기	한 흐름 안의 SIMD — 레인 수는 타입의 일부
access data shared_read . • access out write_only .	읽기만 · 쓰기만 한다는 약속 — 몸을 검사한다	읽기만이면 여러 태스크가 함께 쥔다 · 어기면 E-ACCESS-MODE

표 27.4 — 병렬과 원자 연산의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

parallel <슬라이스> split . 은 되풀이를 나눌 수 있다는 선언이고, 처리기는 자기 몫만 읽고 쓰며 걸음을 넘어 사는 자리에 쓰지 않는지 확인한다. 누적은 reduce <자리> <연산> . 으로 밝히고 연산은 결합적이어야 한다. 나누어 돈 답은 순차와 비트까지 같다. 공유 자리를 함께 갱신할 때는 cap atomic 을 받아 원자 연산을 쓰고, order 로 기억 차례를 고르되 기본은 seq_cst 이며 뜻 없는 조합은 거절된다.

제8부 — 바깥 세계

28 입출력과 파일

이 장의 차례

28.1 표준출력 348
28.2 파일을 통째로 읽는다 349
28.3 열었으면 닫는다 350
28.4 끝과 실패는 다른 답이다 351
28.5 실패를 일부러 일으킨다 352
28.6 흔한 실수 353
28.7 이 장의 문법 한눈에 356

먼저 알아야 할 것

16장 권한 · `cap io·cap file_system` 과 `effects io`
19장 소유 · 완결이 필요한 값은 적어서 끝낸다
17장 실패를 설계하기 · 경계에서는 `result` 로 말한다

돌아보기

19장에서 어떤 타입이 "완결이 필요하다" 는 것은 무엇으로 정해졌는가?

> **답.** 그 타입을 `owned` 로 받아 `result` 를 돌려주는 op 이 있으면 정해졌다. 그런 값을 끝내지 않고 범위를 벗어나게 두면 `E-OWN-INCOMPLETE` 로 거절되었다. 이 장에서 그 규칙이 가장 쓸모 있는 자리 — 열었으면 반드시 닫아야 하는 파일 — 을 본다.

이 장의 필요성과 맥락

제8부는 프로그램이 바깥 세계와 닿는 자리들이다. 첫째가 입출력이다. 입출력은 실패가 흔한 자리다 — 파일이 없고, 디스크가 가득 차고, 읽다가 끊긴다. 그리고 가장 흔한 결함은 실패를 **끝**과 섞는 것, 닫기를 잊는 것이다. Lowent 는 권한으로 누가 입출력을 할 수 있는지 정하고, 소유로 닫기를 잊지 못하게 하며, 답의 모양으로 끝과 실패를 가른다. 앞의 네 부에서 세운 도구들이 한자리에 모인다.

이 장이 끝나면

표준출력에 쓰는 `write_out` 과 표준 라이브러리 `files` 로 파일을 여닫고 읽고 쓰는 법을 익힌다. 파일 핸들이 완결이 필요한 소유 값이라서 닫기를 잊으면 번역이 거절된다는 것을 확인한다. 읽기의 답이 "읽었다·끝이다·실패다" 세 자리로 갈리는 까닭과, 실패를 일부러 일으켜 시험하는 방법도 알게 된다.

이 장에서 답할 질문

1 버퍼보다 큰 파일은 어떻게 읽는가?

28.1 표준출력

표준출력에 쓰는 통로는 `write_out <cap io> <fd> <바이트>` 하나다. 파일 기술자 1 은 표준출력, 2 는 표준오류다. 쓴 바이트 수를 돌려준다. 이미 여러 장에서 썼으므로 새로울 것은 없지만, 한 가지를 짚는다. 이

op 이 권한을 **첫 피연산자**로 받는다는 것은, 표준출력에 한 바이트라도 내는 모든 op 의 머리에 `cap io` 가 보인다는 뜻이다. 디버그용으로 몰래 찍는 출력이 없다.

28.2 파일을 통째로 읽는다

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
```

실행 결과

```
$ lowentc --run main lines.low
first line
second line
third line
main() = 3
```

- `main` 은 권한 셋을 받는다. 출력(`cap io`), 파일(`cap file_system`), 버퍼를 받을 할당(`cap allocator`)이다. 머리만 보고도 이 프로그램이 네트워크에 닿지 않는다는 것을 안다.
- `files.slurp fs "notes.txt" buf` 는 파일 전체를 `buf` 에 읽고 읽은 바이트 수를 `result u64 files.file_error` 로 준다. 파일이 없거나 버퍼보다 크면 오류다. 자르지 않는다.
- 오류면 메시지를 쓰고 종료 코드 2 를 돌려준다. 경계에서 실패를 다루는 모양이다(17장).
- 읽은 부분을 `subslice` 로 잘라 `pipe` 로 줄바꿈을 세고(24장), 내용을 그대로 표준출력에 쓴다.

예제는 자기 폴더의 `notes.txt` 를 읽는다. 세 줄이므로 종료 코드는 3 이다.

> **문. 버퍼보다 큰 파일은 어떻게 읽는가?**
>
> **답.** `files.open` 으로 열고 `files.read` 를 되풀이한다. `read` 는 버퍼만큼만 읽으므로 파일 크기와 상관없이 조각으로 읽을 수 있다. 짧게 읽히는 것은 실패가 아니다. 부탁한 것보다 적게 주는 것은 정상이고, 부르는 쪽이 계속 읽어야 한다. `slurp` 이 안에서 그렇게 한다.

34 바이트짜리 `notes.txt` 를 두 길로 읽으면 이렇게 된다(이 판에서 버퍼 크기를 바꿔 가며 실측했다).

```
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` 은 버퍼가 파일보다 한 바이트만 작아도 `buffer_too_small` 로 답한다. 딱 맞는 크기는 된다. `read` 는 버퍼 크기와 상관없이 조각을 주고, 조각을 이어 붙이는 일은 부르는 쪽이 한다.

28.3 열었으면 닫는다

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
```

실행 결과

```
$ lowentc --run main copy.low
wrote and closed
main() = 0
```

- `files.open fs <경로> 1` 은 쓰기 모드로 연다. 모드는 0 읽기, 1 쓰기(자르고 만들기), 2 덧붙이기다.
- 성공하면 `ok_value o` 를 `owned files.handle` 에 담는다.
- `files.write` 로 쓰고 `files.close fs h` 로 닫는다. `close` 는 `owned handle` 을 받고 `result` 를 돌려준다. 닫기는 진짜로 실패할 수 있다 — 네트워크 파일 시스템이나 가득 찬 디스크에서 마지막 버퍼를 비우지 못하면 닫기에서 드러난다.

`close` 가 그 모양이므로 `handle` 은 완결이 필요한 타입이다. 닫지 않고 두면 거절된다.

examples/ch28/forgot.low

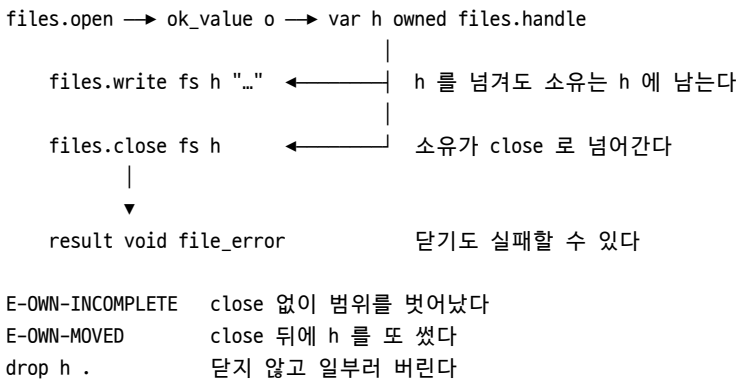
```
module forgot .
rem expect: E-OWN-INCOMPLETE

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
```

실행 결과

```
$ 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
```

닫기를 잊은 프로그램은 번역되지 않는다. 운영체제가 프로세스 종료 때 파일 기술자를 닫아 준다는 기대에 기대지 않는다. 정말로 닫지 않고 버리려면 `drop h .` 로 버린다고 적는다.

핸들 하나의 일생을 한 장으로 그리면 이렇다.



앞의 둘은 번역이 거절한다. 셋째는 버린다는 뜻을 소스에 남기는 길이다.

> **흔한 오해. 파일 핸들을 정수로 들고 다니면 더 가볍다**
>
> 커널이 준 정수를 그대로 들면 실행 비용은 같다. `files.handle` 도 안에 정수 하나뿐인 구조체다. 차이는 번역이 무엇을 알 수 있느냐다. 정수는 복사해도, 잊어도, 두 번 닫아도 아무도 모른다. 소유 값은 옮겨지고, 잊으면 거절되고, 두 번 닫으면 거절된다. 같은 바이트에 규칙을 입힌 것이다.

28.4 끝과 실패는 다른 답이다

`files.read` 의 답은 세 자리다.

답	뜻
`ok (some n)`	`n` 바이트를 읽었다

`ok none`	파일의 **끝**이다. 실패가 아니다
`error e`	실패다. `e` 는 어느 연산이 실패했는지 말한다

표 28.1 — `read` 의 답

34 바이트 파일을 16 바이트 버퍼로 되풀이해 읽으면 답이 이렇게 온다(이 판에서 실측).

```
notes.txt  34 바이트          buf  16 바이트

읽기 1   ok (some 16)    바이트  0 ~ 15
읽기 2   ok (some 16)    바이트 16 ~ 31
읽기 3   ok (some 2)     바이트 32 ~ 33
읽기 4   ok none         끝 — 여기서 멈춘다
읽기 5   ok none         끝에서 또 읽어도 끝이다
```

마지막 조각(2 바이트)이 버퍼보다 작다고 해서 그것이 끝을 알리는 것은 아니다. 끝은 `ok none` 이 따로 알린다.

한때 이 모듈의 `slurp` 은 읽기가 실패하면 반복을 멈추기만 했다. 그러면 실패한 읽기가 "파일을 다 읽었다" 로 보고되었고, 이 모듈로 지은 줄 세기 프로그램이 읽기 실패를 "0 줄" 이라는 성공으로 냈다. 값 하나가 두 뜻(끝과 실패)을 나르면 안 된다는 것이 그때 얻은 교훈이고, 그래서 답이 세 자리가 되었다.

반대로 모든 op 에 세 자리를 기계적으로 씌우지도 않았다. 여는 일에는 "끝" 이 없으므로 `open` 은 `result handle file_error` 두 자리다. 메모리 위의 슬라이스를 읽는 `io` 모듈의 리더는 실패할 자리가 없으므로 `option` 만 쓴다. op 마다 정직한 모양이 다르다.

28.5 실패를 일부러 일으킨다

파일 실패는 평소에 잘 일어나지 않아서, 실패를 다루는 코드는 시험되지 않은 채 남기 쉽다. 표준 라이브러리의 호스트 연산에는 결함 주입기가 있다. 환경 변수로 켠다.

```
LOW_HOST_FAULT="open:err"        모든 open 이 실패한다
LOW_HOST_FAULT="read:err@2"      두 번째 읽기가 실패한다
LOW_HOST_FAULT="read:short@1=4"  첫 읽기를 4 바이트로 자른다
LOW_HOST_FAULT="close:err"       닫기가 실패한다
```

VM 과 네이티브가 같은 주입기를 쓰므로 두 백엔드의 답이 같아야 한다. `LOW_HOST_FAULT="open:err" lowentc --run main lines.low` 로 돌리면 `lines.low` 는 오류 갈래로 가서 메시지를 쓰고 2 를 돌려준다. 기본은 꺼져 있다.

앞 절의 16 바이트 읽기를 주입기 아래에서 다시 돌리면 답이 이렇게 바뀐다(이 판에서 실측).

```
(주입 없음)        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
```

짧게 읽혀도 조각의 합은 34 로 같고, 끝은 여전히 `ok none` 이 알린다. `eof_fixed.low` 는 첫 줄과 둘째 줄 모두에서 34 를 돌려준다. 셋째 줄에서는 실패가 끝과 다른 답으로 오므로, 둘을 가르는 코드만 이 경로를 바르게 지난다.

> **실제 사례. 줄 세기 프로그램이 찾아낸 결함**
>
> 위의 "0 줄" 결함은 주입기가 실물로 보였다. `LOW_HOST_FAULT="read:err@1"` 로 첫 읽기를 실패시키자 줄 세기가 성공으로 끝났다. 실패 경로가 한 번도 돌지 않았다면 이 결함은 실제 디스크 오류가 날 때까지 숨어 있었을 것이다. 권한과 소유가 "닫기를 잊음" 을 번역에서 막는다면, 주입기는 "실패를 잘못 다룸" 을 실행에서 드러낸다.

28.6 흔한 실수

반례. 읽기 모드로 열고 쓴다 — 쓴 바이트 수를 보지 않는다

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 ✘ 모드 0 은 읽기다 --- 그런데 쓴다
  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 쓰기가 실패했으므로 여기서 2 로 나간다 --- 짧게 쓰인 것도 함께 본다
  guard eq (ok_value w) 11 . else return 3 .
  return 0 .
end
```

실행 결과

```
$ lowentc --run main mistake_readmode.low
main() = 2
```

`files.open fs "notes.txt" 0` 은 읽기 모드다. 거기에 쓰면 `files.write` 가 실패를 돌려주고(스트림 오류가 켜진 짧은 쓰기는 값이 아니라 실패다) 이 예제는 종료 코드 2 로 나간다. 전에는 같은 자리가 `ok 0`("0 바이트 썼다")이어서 `is_ok w` 만 본 코드가 성공으로 지나갔다. 모드를 확인하고(0 읽기 · 1 쓰기 · 2 덧붙이기), 쓰기의 답은 성공 여부와 함께 **쓴 수**까지 본다. 짧게 쓰인 것은 남은 바이트를 다시 써야 한다는 뜻이다.

반례. 닫은 핸들로 또 쓴다

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 ✘ 닫은 핸들로 또 쓴다 --- `close` 가 소유를 가져갔다
```

```
  let w result u64 files.file_error be files.write fs h "late\n" .
  drop w .
  return 0 .
end
```

실행 결과

```
$ 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` 는 `owned handle` 을 받으므로 부르는 순간 소유가 넘어간다. 그 뒤의 `h` 는 이미 없는 핸들이라 `E-OWN-MOVED` 다. C 에서 `fclose` 뒤의 `fwrite` 는 정의되지 않은 동작이고, 같은 번호가 새로 연 다른 파일에 재사용되었다면 엉뚱한 파일에 쓴다. 소유가 그 자리를 번역에서 막는다.

반례. 읽기의 답에서 실패만 묻는다

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 ✘ 실패만 물었다 --- 파일 끝의 `ok none` 에서 `some_value` 가 멈춘다
    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
```

실행 결과

```
$ lowentc --run main mistake_eof.low
== ir diagnostics (1) ==
0:0 E-VM-NONE: some_value of none (panic)
```

`files.read` 의 답은 세 자리다. `is_ok r` 는 "실패가 아니다" 만 말한다. 파일 끝의 `ok none` 도 실패가 아니므로 통과하고, 그 안에서 `some_value` 를 꺼내다 멈춘다. 세 자리를 모두 가른다.

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 세 자리를 모두 가른다 --- 읽었다 · 끝이다 · 실패다
    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
```

실행 결과

```
$ lowentc --run main eof_fixed.low
main() = 34
```

16 바이트 버퍼로 34 바이트 파일을 세 번에 나누어 읽고, 넷째 읽기의 `ok none` 에서 멈춘다. 실패하면 닫고 3 을, 닫기가 실패하면 4 를 돌려준다. `rounds` 의 상한은 끝을 영영 알리지 않는 원천에서도 반복이 끝나게 한다.

흔한 오해. 버퍼보다 큰 파일을 slurp 하면 앞부분만 읽힌다

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 34 바이트 파일에 8 바이트 버퍼
  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 앞 8 바이트로 잘려 오지 않는다 --- 오류다
  guard not (is_error r) . else return 2 .
  return narrow u8 (ok_value r) .
end
```

실행 결과

```
$ lowentc --run main slurp_small.low
main() = 2
```

8 바이트 버퍼에 34 바이트 파일을 slurp 하면 앞 8 바이트가 오는 것이 아니라 오류가 온다. 잘린 내용을 온전한 파일로 믿는 결함 — 설정 파일의 뒷부분이 조용히 사라지는 일 — 을 막으려는 것이다. 파일이 버퍼보다 클 수 있으면 files.open 과 files.read 로 조각씩 읽는다.

28.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`write_out out 1 "…"`	표준출력(1)·표준오류(2)에 쓴다 — 쓴 수를 준다	권한이 첫 피연산자 — 몰래 찍는 출력이 없다
`use files . + input fs cap file_system .`	파일 모듈과 그 권한	머리만 보고 파일에 닿는지 안다
`files.open fs "notes.txt" 0`	연다 — 0 읽기 · 1 쓰기 · 2 덧붙이기 · result handle file_error	여는 일에는 "끝" 이 없다 — 두 자리
`var h owned files.handle be ok_value o .`	핸들을 소유로 담는다	잊으면 E-OWN-INCOMPLETE · 닫은 뒤 쓰면 E-OWN-MOVED
`files.read fs h buf`	ok (some n) 읽었다 · ok none 끝 · error e 실패	값 하나가 두 뜻을 나르지 않는다
`files.write fs h bytes`	쓴 수를 result 로 준다	짧게 쓰일 수 있다 — 수를 확인한다
`files.close fs h`	owned 로 받아 result — 완결	닫기도 실패할 수 있다

`files.slurp fs path buf`	통째로 읽는다 — 버퍼보다 크면 오류	자르지 않는다
`LOW_HOST_FAULT="read:err@2"`	실패를 일부러 일으킨다(환경 변수)	실패 경로를 시험한다

표 28.2 — 입출력과 파일의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

표준출력은 `write_out <cap io> <fd> <바이트>` 이고 권한이 첫 피연산자다. `files` 모듈은 `cap file_system` 을 받아 파일을 여닫고 읽고 쓴다. 파일 핸들은 `close` 가 `owned` 로 받아 `result` 를 돌려주므로 완결이 필요한 값이고, 닫기를 잊으면 번역이 거절한다. `read` 의 답은 읽었다·끝이다·실패다 세 자리이며, 실패 경로는 `LOW_HOST_FAULT` 주입기로 일부러 돌려 본다.

29 C 와 만나는 자리

이 장의 차례

29.1 C 를 부른다 — 표시·권리·효과 줄 358
29.2 경계를 건너는 타입 360
29.3 C 가 Lowent 를 부른다 361
29.4 되부름과 소유 362
29.5 흔한 실수 364
29.6 이 장의 문법 한눈에 367

먼저 알아야 할 것

16장 권한 · `cap c` 는 시작점이 받을 수 없는 권한이다
14장 계약 · `requires` 는 진입에서 검사된다
21장 모듈 · `export` 한 op 만 C 에서 부를 수 있는 심볼이 된다

돌아보기

16장에서 `cap c` 는 왜 시작점이 요구할 수 없는 권한이라고 했는가?

> **답.** 실행하는 쪽(운영체제)이 건넬 수 있는 권한이 아니기 때문이다. C 로 들어가는 문은 그것을 줄 자격이 있는 자리에서 만들어져 인자로 흘러야 한다. 이 장은 그 문 — `extern` — 을 다룬다. 들어가는 방향과 나가는 방향 모두다.

이 장의 필요성과 맥락

새 언어가 현실에서 쓰이려면 이미 있는 C 코드와 만나야 한다. 운영체제의 API, 수십 년 된 라이브러리, 하드웨어 공급사의 SDK 가 모두 C 다. 그런데 C 로 넘어가는 순간 이 언어가 지키던 모든 것 — 경계 검사, 소유, 효과 — 이 C 쪽에서 깨질 수 있다. Lowent 는 이 자리를 편하게 만드는 대신 **좁고 보이게** 만든다. 그리고 반대 방향, C 가 Lowent 를 부르는 자리에는 계약을 세워 문 안을 지킨다.

이 장이 끝나면

C 함수를 부르는 `extern` op 이 갖춰야 할 세 가지(`unsafe` 표시·`cap c`·효과 줄)와 `link` 절을 익힌다. 경계를 건널 수 있는 타입이 C ABI 가 표현할 수 있는 것으로 한정되고 슬라이스는 포인터와 길이 둘이 된다는 것을 알게 된다. `export` 한 op 을 `--emit-h`·`--no-main` 으로 C 프로그램에 넣고, C 가 계약을 어기면 문에서 멈추는 모습을 확인한다. 되부름과 소유를 넘기는 규칙도 보게 된다.

이 장에서 답할 질문

1 `unsafe` 가 붙은 op 안에서는 아무 일이나 해도 되는가?

29.1 C 를 부른다 — 표시·권리·효과 줄

`examples/ch29/area.low`

```
module area_ffi .

rem 몸이 C 에 있다 — 표시·권리·효과 줄 셋을 모두 갖춘다
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
```

실행 결과

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

C 함수를 부르는 op 은 셋을 **모두** 갖춘다.

갖출 것	없으면	누구에게 말하나
`unsafe` **표시**	`E-FFI-NOUNSAFE`	사람에게 — 여기서부터는 언어가 아니라 사람이 책임진다
`input k cap c .`	`E-FFI-NOCAP`	처리기에게 — C 로 들어가는 것은 건네받은 권리다
효과 줄(적어도 `effects unsafe`**)**	`E-FFI-NOEFFECT`	부르는 쪽에게 — 무엇을 떠안는지 머리에서 배운다

표 29.1 — C 를 부르는 op 이 갖추어야 하는 것

`extern` op 은 몸이 C 에 있으므로 이쪽에는 몸이 없다. 대신 `struct` 가 칸을 담듯 **절을** `do … end` 에 담고, 그 가운데 `link "lw_c_area"` 가 C 쪽 이름을 댄다. 블록에 문장을 적으면 몸이 둘이 되어 `E-FFI-BODY` 다. 이름은 처리기가 op 이름에서 지어내지 않는다. 다른 언어에 하는 약속이므로 약속한 사람이 적는다 — 빠뜨리면 `E-FFI-LINK` 다. `area_twice` 는 그 op 을 부르므로 스스로도 `unsafe` 이고 `cap c` 를 받는다. 표시와 권리가 호출 사슬을 따라 올라간다.

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
```

실행 결과

```
$ 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
```

실행 결과

```
$ 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
```

셋 중 하나만 있어도 "부를 수는 있다". 그러나 그러면 이 언어는 자기가 무엇을 잃었는지 말하지 못한다.

> **문.** `unsafe` **가 붙은 op 안에서는 아무 일이나 해도 되는가?**
>
> **답.** 아니다. `unsafe` 는 "아무렇게나 해도 된다" 가 아니라 "이 자리의 일부를 처리기가 검사할 수 없다" 는 표시다. `unsafe` op 의 본문에서도 경계 검사·소유·차용 규칙은 그대로 적용된다. 검사할 수 없는 것은 C 함수의 **안**이다. 그 표시가 소스에 남아 있어서 나중에 감사할 자리를 찾을 수 있다.

29.2 경계를 건너는 타입

경계를 건널 수 있는 것은 C ABI 가 표현할 수 있는 것뿐이다. `option`·`result`·벡터·그릇은 C 에 없고, 처리기는 있는 척하지 않는다.

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
```

실행 결과

```
$ 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)
```

저절로 사상되는 것은 **슬라이스뿐**이다. `slice t` 는 "`t` 를 가리키는 포인터" 와 "개수" 두 인자가 된다. 폭도 안다 — `slice u32` 는 `uint32_t *` 이지 바이트 포인터가 아니다. 권한은 값이 아니므로 C 로 건너가지 않는다. ABI 이름은 `c` 하나다. "이 기계에서 C 가 무엇인가" 는 짓는 기계가 이미 정한다.

29.3 C 가 Lowent 를 부른다

반대 방향이다. `export` 한 op 은 C 에서 부를 수 있는 심볼이 된다.

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
```

실행 결과

```
$ lowentc --run clamp_add exported.low 100 50
clamp_add(100, 50) = 150
```

`--emit-h` 가 헤더를 낸다. 헤더를 손으로 적으면 서명이 두 곳에 살고, 언젠가 갈린다.

```
long long clamp_add(long long, long long);
long long sum_bytes(const unsigned char *, size_t);
```

`sum_bytes` 의 `slice u8` 이 포인터와 길이 둘로 나뉜 것을 볼 수 있다.

```
Lowent
  input xs slice u8 .

C (--emit-h 가 낸 머리)
```

```
   sum_bytes(const unsigned char *, size_t)
             └──────────┬──────────┘ └──┬──┘
                        │               개수 (len xs)
                        포인터 (xs 의 첫 칸)
```

`--no-main` 은 `main` 과 명령 줄 디스패처 없이 내보낸 op 의 진입점만 담은 C 를 낸다. 남의 빌드에 그대로 넣으면 된다. 다음 C 프로그램이 그 둘을 쓴다.

```c
// 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)
```

앞의 두 호출은 답을 받는다. 셋째 호출은 `requires le a 1000 .` 을 어긴다. C 는 계약을 모르지만, **불려 들어오는 자리는 이쪽 문**이므로 op 의 계약이 인자에 강제되고 진입에서 멈춘다. 문 안은 Lowent 가, 문 밖은 C 가 책임진다.

두 방향을 한 장에 그리면 이렇다. 나가는 쪽은 **머리에 적고**, 들어오는 쪽은 **문에서 잰다**.

```
나가는 쪽 --- Lowent 가 C 를 부른다
  area_twice ──▶ c_area ═ link "lw_c_area" ═▶  lw_c_area (C)
  unsafe · cap c · effects unsafe              안은 검사 밖, 사람이 책임진다

들어오는 쪽 --- C 가 Lowent 를 부른다
  host.c ┄ clamp_add(5000, 1) ═▶ | requires le a 1000 .
                                 | 5000 은 약속을 어긴다
                                 └▶ 진입에서 멈춘다 (종료 코드 70)
```

> **흔한 오해. FFI 경계에서는 언어의 보장이 모두 사라진다**
>
> C 로 **나가는** 쪽에서는 C 함수 안이 검사 밖이다. 그러나 그 사실은 `unsafe·cap c·효과` 줄로 머리에 적힌다. C 가 **들어오는** 쪽에서는 보장이 그대로 선다. 들어오는 인자가 계약을 어기면 진입에서 멈춘다. 경계는 구멍이 아니라 문이고, 문에는 계약이 서 있다.

29.4 되부름과 소유

C 에게 Lowent 함수를 넘겨 되부르게 하려면 `export extern` op 의 주소를 `unsafe_fn <op>` 으로 얻어 값으로 넘긴다. 새 규약은 없다. 규칙이 둘 붙는다.

- `unsafe_fn` 이 가리킬 수 있는 것은 `export extern` op 뿐이다(`E-FN-NOTEXPORT`). C 가 부를 수 없는 문의 주소는 주소가 아니다.
- 되부름으로 쓸 op 은 권한을 요구할 수 없다(`E-FN-CAP`). 되부름에 들어서는 것은 C 이고, C 는 건넬 권한이 없다. 권한이 필요한 일은 되부름 밖에서 하고, 되부름 안은 셈만 한다.

`owned τ` 를 `extern` 에 넘기면 없앨 책임이 C 에게 옮겨간다. 이쪽의 의무는 그 지점에서 끝나고, 그 뒤의 반납이 지켜졌는지는 검증되지 않는다. 고정된 인자 뒤에 개수가 정해지지 않은 인자를 받는 C 함수는 `variadic .` 절로 부를 수 있지만, 그 인자에는 계약이 닿지 않으며 반대 방향(C 가 우리 가변인자를 부르기)은 없다.

examples/ch29/variadic.low

```
module variadic .

rem C 의 char* — unsafe_ptr 는 타입 앞에 붙는 한정자다
newtype cstr unsafe_ptr u8 .

unsafe extern proc c_printf do
  input k cap c .
  input fmt cstr .
  output i32 .
  effects unsafe .
  rem 고정 인자 뒤에 개수가 정해지지 않은 인자를 받는다
  variadic .
  link printf .
end

rem 고정 인자(서식) 뒤에 42 를 하나 더 건넨다 — 그 값에는 타입 검사도 계약도 닿지 않는다
unsafe proc report input k cap c . output i32 . effects unsafe .
do
  return c_printf k (cstr_of "sum=%d\n\0") 42 .
end
```

실행 결과

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

- `newtype cstr unsafe_ptr u8 .` 는 C 의 `char*` 를 이름 붙인 타입으로 만든다. `unsafe_ptr` 는 홀로 쓰는 타입이 아니라 타입 앞에 붙는 한정자다.
- `variadic .` 절이 "고정 인자 뒤에 더 받는다" 를 적고, `link printf .` 가 C 이름을 적는다. 부르는 자리의 `42` 는 고정 인자 뒤에 붙는 값이라 타입 검사도 계약도 닿지 않는다. 서식과 인자가 어긋나는 C 의 고전적 결함이 여기서는 막히지 않는다는 뜻이다. 그래서 `unsafe` 가 붙는다.
- `cstr_of "…\0"` 은 끝에 영 바이트를 둔 문자열 리터럴을 C 문자열로 본다. VM 은 C 를 부를 수 없으므로 이 예제는 검사만 하고, 네이티브로 지으면 `sum=42` 를 찍는다.

실제 사례. 두 수를 더하는 프로그램이 지는 짐

기본 방출은 **프로그램**이다. `main` 과 명령 줄 디스패처가 함께 나오고, 디스패처는 태그 경로와 그 풀들을 붙잡는다. 개발 저장소의 측정에서 두 수를 더하는 op 하나를 기본으로 내면 읽기 전용 데이터가 약 197 KB, 초기화되지 않은 데이터가 약 1.7 MB 였고, `--no-main` 으로 내면 둘 다 50 바이트 이하가 되었다. 라이브러리로 넣을 때 `--no-main` 을 쓰는 이유다. 운영체제 없는 대상(`--target cortex_m`)은 처음부터 디스패처를 내지 않는다.

29.5 흔한 실수

반례. `extern` op 에 효과 줄을 적지 않는다

examples/ch29/mistake_noeffect.low

```
module mistake_noeffect .
rem expect: E-FFI-NOEFFECT

rem ✘ 효과 줄이 없다 --- 부르는 쪽이 무엇을 떠안는지 머리에서 알 수 없다
unsafe extern proc c_area do
  input k cap c .
  input w i64 .
  input h i64 .
  output i64 .
  link "lw_c_area" .
end
```

실행 결과

```
$ 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`)
```

진단이 둘 나온다. `E-FFI-NOEFFECT` 는 "C 를 부르는데 효과 줄이 없다" 이고, `E-UNSAFE-UNUSED` 는 "`unsafe` 표시를 했는데 `unsafe` 효과가 없으니 거짓 경보다" 이다. 둘은 같은 뿌리에서 나온다. 표시(`unsafe`)는 **누가 책임지는가**, 효과 줄(`effects unsafe`)은 **무엇을 하는가** 를 말하고, 한쪽만 있으면 짝이 맞지 않는다. `effects unsafe .` 한 줄로 둘 다 사라진다.

반례. `extern` op 에 `link` 절을 적지 않는다

examples/ch29/mistake_nolink.low

```
module mistake_nolink .
rem expect: E-FFI-LINK

rem ✘ `link` 절이 없다 --- 어느 C 심볼을 부르는지 아무 데도 안 적혀 있다
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
```

실행 결과

```
$ 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)
```

`link` 가 없으면 어느 C 심볼을 부르는지 소스 어디에도 없다. 처리기가 op 이름에서 지어내면 op 의 이름을 바꾸는 순간 **다른 C 함수**를 부르게 되고 그 사실이 적힌 데가 없다. 그래서 `E-FFI-LINK` 로 거절한다 — 약속은 적은 사람의 것이다.

반례. C 를 부르는 op 을 부르면서 `unsafe` 표시를 빠뜨린다

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 ✘ C 를 부르는 op 을 부르면서 `unsafe` 표시를 빠뜨렸다
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
```

실행 결과

```
$ 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` 는 `effects unsafe` 를 적었지만 머리에 `unsafe` 가 없다. 효과는 호출을 따라 올라가므로 부르는 op 도 `unsafe` 효과를 내고, 그 효과를 낸다면 누군가 서명해야 한다. `E-UNSAFE-UNDECLARED` 의 말대로 "아무도 서명하지 않은 unsafe op 이 바로 이 언어가 막으려는 구멍" 이다. `unsafe proc area_twice …` 로 적는다.

반례. 보통 op 의 주소를 되부름으로 넘긴다

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 ✘ 보통 op 의 주소를 C 에게 넘기려 한다 --- C 가 부를 수 있는 문이 없다
  let f u64 be unsafe_fn by_value .
  return f .
end
```

실행 결과

```
$ 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 가 부를 수 있는 것은 `export extern op` 이 만든 심볼뿐이다. 그 입구에서 계약이 인자에 강제된다. 보통 op 에는 그런 문이 없으므로 `E-FN-NOTEXPORT` 다. 머리를 `export extern fn by_value …` 로 바꾼다. 첫 줄의 `W-EFFECT-OVER` 는 주소를 얻는 것만으로는 `unsafe` 일을 한 것이 아니라는 알림이다. 그 주소로 C 를 부르는 op 에서 `unsafe` 가 선다.

반례. 되부름으로 쓸 op 이 권한을 받는다

examples/ch29/mistake_cbcap.low

```
module mistake_cbcap .
rem expect: E-FN-CAP

rem 되부름으로 쓸 op 이 권한을 입력으로 받는다
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 ✘ C 가 되부를 때 건넬 권한이 없다
  let f u64 be unsafe_fn cmp_logged .
```

```
  return f .
end
```

실행 결과

```
$ 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)
```

되부름에 들어서는 것은 C 이고 C 에게는 건넬 권한이 없다. 그러니 권한을 받는 op 은 되부름이 될 수 없다(E-FN-CAP). 출력이나 파일처럼 권한이 필요한 일은 되부름 **밖**에서, C 가 돌아온 뒤에 한다. 되부름 안은 셈만 한다.

29.6 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`unsafe extern proc c_area do input k cap c . … effects unsafe . link "lw_c_area" . end`	C 에 몸이 있는 op — 절을 `do … end` 에 담는다	표시·권리·효과 줄 셋이 모두 있어야 한다
`unsafe proc area_twice input k cap c . … effects unsafe .`	C 를 부르는 op 을 부르는 op	표시와 권리가 호출 사슬을 따라 올라간다
`input xs slice u8 .` **(경계)**	C 에서는 포인터와 길이 두 인자	저절로 사상되는 것은 슬라이스 뿐
`option`·`result`·**벡터를 경계에**	거절(E-FFI-TYPE)	C ABI 에 없는 것을 있는 척하지 않는다
`export fn clamp_add …`	C 에서 부를 수 있는 심볼	들어오는 인자에 계약이 강제된다
`lowentc --emit-h` · `--no-main`	헤더를 낸다 · `main` 없이 라이브러리로 낸다	서명이 한 곳에만 산다
`unsafe_fn cmp`	`export extern` op 의 주소(되부름)	보통 op 은 E-FN-NOTEXPORT · 권한을 받으면 E-FN-CAP
`input h owned τ .` **(extern 에)**	없앨 책임이 C 로 넘어간다	그 뒤의 반납은 검증되지 않는다
`variadic .`	C 의 가변 인자 함수를 부른다	가변 인자에는 계약이 닿지 않는다
`newtype cstr unsafe_ptr u8 .` · `cstr_of "…\0"`	C 포인터 한정자 · 영 바이트로 끝나는 C 문자열로 보기	포인터는 이름 붙인 타입으로만 다룬다

표 29.2 — C 경계의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

C 를 부르는 `extern` op 은 `unsafe·cap c·`효과 줄을 모두 갖추고 `link` 로 C 이름을 댄다. 경계를 건너는 타입은 C ABI 가 표현할 수 있는 것뿐이고 슬라이스는 포인터와 길이가 된다. `export` 한 op 은 `--emit-h·--no-main` 으로 C 에 넣으며, C 가 계약을 어기면 문에서 멈춘다. 되부름은 권한 없는 `export extern` op 의 주소로 하고, `owned` 를 넘기면 책임이 C 로 간다.

30 하드웨어 — 레지스터, 인터럽트, 기계 명령

이 장의 차례

30.1 레지스터 지도 370
30.2 닿는 법은 번역에서 강제된다 371
30.3 인터럽트 처리기 372
30.4 기계가 감당하는 것 — `build tier` 373
30.5 기계 명령 — `asm` 374
30.6 흡수 경계 — `unsafe` 가 멈추는 자리 375
30.7 기계가 가진 것을 쓸까 — `--hw` 376
30.8 흔한 실수 377
30.9 이 장의 문법 한눈에 380

먼저 알아야 할 것

13장 이름 붙인 타입 · `layout`·`view` 와 칸의 닿는 법 표지
20장 할당기와 고정 메모리 · 운영체제 없는 기계의 고정 창
29장 C 와 만나는 자리 · 검사할 수 없는 자리는 표시·권리·효과 줄로 가둔다

돌아보기

29장에서 C 를 부르는 op 이 갖춰야 하는 셋은 무엇이었고, 각각 누구에게 말하는 것이었는가?

> **답.** `unsafe` 표시(사람에게), `cap c`(처리기에게), 효과 줄(부르는 쪽에게)이었다. 이 장의 장치 레지스터와 기계 명령도 같은 방식으로 가둔다. 권한의 이름과 효과의 이름만 달라진다.

이 장의 필요성과 맥락

시스템 언어의 가장 낮은 자리는 운영체제 없이 장치 레지스터를 직접 읽고 쓰는 펌웨어다. 이 자리의 결함은 조용하고 치명적이다. 컴파일러가 "쓸모없어 보이는" 레지스터 읽기를 지우거나, 쓰기 전용 레지스터를 읽어 장치를 움직이거나, 인터럽트 처리기를 보통 함수처럼 불러 틀린 스택에서 돌린다. Lowent 는 이것들을 번역에서 막는다. 앞의 장들에서 세운 도구 — 권한·효과·배치· 고정 창 — 이 하드웨어에서 어떻게 쓰이는지 모은다.

이 장이 끝나면

`mmio <주소>` 를 붙인 구조체로 장치의 레지스터 지도를 적고, `read_volatile`·`write_volatile` 로 닿는 법을 익힌다. `cap mmio` 와 `device` 효과, 읽기 전용·쓰기 전용 표지가 번역에서 강제된다는 것, 장치를 값으로 받을 수 없는 이유를 알게 된다. `vector` 절로 인터럽트 처리기를 만들고, `build tier` 로 기계가 감당하는 효과를 정하며, `asm` 절로 기계 명령을 가두어 쓰는 법도 보게 된다.

이 장에서 답할 질문

1 C 에서는 `volatile` 한정자를 붙인다. 무엇이 다른가?

30.1 레지스터 지도

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
```

실행 결과

```
$ 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` 는 장치의 **지도**다. `mmio` 뒤의 수가 시작 주소이고, 칸들이 차례로 레지스터가 된다. 새 낱말은 없다. 구조체에 절 하나가 붙었을 뿐이다.
- 칸 뒤의 `rw·ro·wo` 가 닿는 법이다.
- `read_volatile·write_volatile` 이 레지스터에 닿는다. 처리기는 이 접근을 **합치거나 없애거나 차례를 바꾸지 않는다.** 장치 레지스터는 읽는 행위 자체가 일이라서, 읽으면 상태가 바뀌기도 한다.
- 레지스터에 닿으려면 `cap mmio` 를 받고 `effects device` 를 적는다. 권한 없는 하드웨어 접근은 할당기에서 본 주변 권한과 정확히 같은 문제다.
- `build tier t1 .` 은 이 모듈이 작은 기계용이라는 선언이다(아래에서 다룬다).

VM 에서는 이것이 **실제로 돈다**. 장치 대신 호출자가 건넨 바이트 버퍼가 레지스터 자리가 되고, `view gpio regs` 가 그 위에 지도를 얹는다. 실행 결과의 인자 모습 `[2,0,0,0,7,0,0,0,1,0,0,0]` 에서 `moder` 에 2, `bsrr` 에 1 이 쓰였고 `idr` 의 7 이 읽혀 돌아온 것을 볼 수 있다. 실행 인자의 첫 `0` 은 `cap mmio` 자리를 채우는 자리표다.

지도와 버퍼를 겹쳐 그리면 이렇다. 칸이 선언 차례로 4 바이트씩 놓인다.

```
주소        칸     닿는 법  VM 에서 (u32 하나 = 4 바이트)
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
```

> **문. C 에서는 `volatile` 한정자를 붙인다. 무엇이 다른가?**
>
> **답.** C 의 `volatile` 은 변수의 성질이고 잊기 쉽다. 붙이지 않으면 최적화기가 읽기를 지워도 경고가 없다. Lowent 에서는 레지스터에 닿는 일이 `read_volatile`·`write_volatile` 이라는 이름 붙은 **연산**이고, 그 연산을 쓰는 op 의 머리에 `device` 효과와 `cap mmio` 가 선다. 읽기를 지워도 되는지는 변수에 붙은 한정자를 기억하는 일이 아니라 연산의 이름이 정한다.

30.2 닿는 법은 번역에서 강제된다

읽기 전용 레지스터에 쓰면 거절된다.

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
```

실행 결과

```
$ 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
```

쓰기 전용 레지스터를 읽는 것도 같다. 쓰기 전용 레지스터를 읽으면 쓰레기가 나오거나 읽는 행위가 장치를 움직인다. 그 읽기는 쓸모없는 것이 아니라 **틀린** 것이다. 장치가 무엇을 받아들이는지 말했으면 타입이 그것을 되말한다.

장치를 값으로 받을 수도 없다.

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
```

실행 결과

```
$ 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)
```

구조체 매개변수는 복사된다. 장치를 복사한다는 것은 쓰기 전용까지 포함한 모든 레지스터를 한꺼번에 읽는다는 뜻이고, 복사본에 쓰는 일은 장치에 닿지 않는다. 그렇게 쓰인 프로그램은 아무 일도 하지 않으면서 아무 말도 하지 않는다. 그래서 번역에서 막는다. 레지스터 묶음은 슬라이스에 `view` 로 얹어 다룬다. 운영체제가 있는 기계에서 절대 주소를 열려고 하면 `E-MMIO-NOHOST` 로 거절된다 — 그 주소는 장치가 아니다.

30.3 인터럽트 처리기

op 에 `vector <번호> .` 절을 붙이면 그 op 은 인터럽트 처리기가 된다. 급함은 `priority <수> .` 로 적는다.

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
```

실행 결과

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

이 예제는 `--target cortex_m` 으로 검사한다. 인터럽트 처리기는 기계가 부르므로 넷을 지킨다.

지킬 것	어기면	왜
아무도 부르지 않는다	`E-ISR-CALLED`	우리 코드가 부르면 틀린 스택·틀린 우선순위에서 돈다
매개변수가 없다	`E-ISR-PARAMS`	기계는 인자를 건네지 않는다
돌려주는 것이 없다	`E-ISR-OUTPUT`	돌려줄 상대가 없다
`effects device` 를 적는다	`E-ISR-EFFECT`	장치 때문에 있는 자리다

표 30.1 — 인터럽트 처리기가 지켜야 하는 것

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
```

실행 결과

```
$ 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)
```

인터럽트 처리기와 보통 코드가 나누어 쓰는 상태는 우선순위와 큐의 규율로 오간다. 그 규율은 언어가 아니라 라이브러리의 일이다. 락 없이 한 생산자와 한 소비자가 값을 넘기는 `spsc` 링 버퍼가 그 자리에 쓰인다(34장).

30.4 기계가 감당하는 것 — `build tier`

`build tier <이름> .` 은 이 기계가 무엇을 **감당하는가**를 말한다. 등급마다 쓸 수 있는 효과가 정해져 있다.

등급	어떤 기계	새로 감당하는 효과
`t0`	아주 작은 기계, 운영체제 없음	`none unsafe panic state wait cancel`
`t1`	작은 기계	위에 더해 `io device`
`t2`	실시간 운영체제	위에 더해 `alloc lock atomic blocking concurrent`
`t3`	운영체제가 있는 기계	위에 더해 `heap page_fault detach` — 곧 전부

표 30.2 — 등급과 감당하는 효과

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
```

실행 결과

```
$ 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
```

`t1` 기계에서 할당(`alloc`)은 감당하지 못한다고 선언했으므로 거절된다. 이것은 실행 중에 느려지는 문제가 아니라 애초에 그 기계에 실을 수 없는 것이다. 효과는 부르는 사슬을 따라 올라오므로, 거절은 효과가 처음 생긴 자리가 아니라 부르는 자리에서도 난다. 등급을 적지 않으면 `t3` 이고 아무것도 막지 않는다.

등급(무엇을 감당하는가)과 프로파일(25장 — 어떤 동시성 짜임을 쓰는가)은 다른 축이다. 운영체제가 있는 작은 기계도 있고, 없는 큰 기계도 있다.

> **흔한 오해. `--target cortex_m` 을 주면 등급도 저절로 정해진다**
>
> 대상(`--target`)은 기계어와 `machine.*` 상수(힙이 있는가 따위)를 정하고, 그래서 힙은 대상만으로도 거절된다 (18장). 등급은 소스에 적는 **약속**이다. 같은 대상이라도 `t0` 를 적으면 입출력까지 막을 수 있다. 약속을 적은 사람만 그 약속을 지고, 처리기는 적지 않은 약속을 지어내지 않는다.

30.5 기계 명령 — `asm`

특권 명령, 시스템 호출, 정확한 사이클 수처럼 이식 가능한 코드로 닿지 못하는 것이 있다. 그 자리에서 op 의 몸을 기계 명령으로 쓴다.

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
```

실행 결과

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

- `asm x86_64 .` 절이 기계를 정한다. 지금 짓는 기계와 다른 기계의 어셈블리는 거절된다. 처리기는 이식되는 척하지 않는다.
- `reg a .` 는 입력 `a` 를 레지스터에 싣고, `out reg r .` 은 결과를 받는다. `clobber flags .` 는 망가뜨리는 것을, `options …` 는 처리기에게 주는 약속을 적는다.
- 몸은 `text ASM … ASM` 이음글 하나다. 어셈블리와 보통 코드를 섞을 수 없다.
- 가둠은 넷이다 — `unsafe` 표시, `cap machine`, `effects unsafe`, 기계 이름.

처리기는 템플릿 안을 읽지 못한다. 그러나 피연산자 목록과 템플릿의 `{이름}` 은 같은 것을 말하는 두 표현이므로 양쪽으로 맞대어 검사한다.

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
```

실행 결과

```
$ 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
```

선언하지 않은 `{c}` 를 템플릿이 부르면 어셈블러는 그것을 거절하거나, 더 나쁘게는 그 자리에 있던 레지스터를 그냥 읽는다. 반대로 선언한 피연산자를 템플릿이 쓰지 않아도 거절된다(`E-ASM-UNUSED`). `options pure` 는 "곁효과가 없다" 는 약속이라 처리기가 그것을 믿고 호출을 지우거나 합칠 수 있는데, 효과 줄이 입출력이나 장치를 만진다고 말하면 둘 다 참일 수 없으므로 거절된다 (`E-ASM-OPTLIE`). VM 은 기계 명령을 돌리지 못하고 못 돌린다고 말한다(`E-VM-ASM`). 네이티브에서만 돈다.

30.6 흡수 경계 — unsafe 가 멈추는 자리

앞 절의 가둠 넷에는 대가가 하나 딸려 온다. `effects unsafe` 는 **부름을 따라 끝까지 번진다.** 어셈블리 잎을 한 줄 쓰면 그것을 부른 op 도, 그 op 을 부른 op 도 차례로 `unsafe` 를 적어야 한다. 표준 라이브러리에서 기계 명령이 한 번도 안 쓰인 까닭이 그것이다(2026-09-23 실측: `lib/` 예순네 모듈에서 `asm` 사용 0).

번짐 자체는 옳다. 처리기가 못 보는 일을 부르는 쪽이 모르면 효과 줄은 거짓말이 된다. 빠진 것은 **책임을 넘겨받는 자리**였다.

```
export proc add2 input a u64 . input b u64 . output u64 . effects none .
  absorbs machine k .          rem 여기서 멈춘다. `k` 가 이 몸 안의 `cap machine` 이다
  reference add2_soft .        rem 같은 답을 내야 하는 순수한 판
  why "레지스터 둘을 더할 뿐 메모리를 만지지 않는다(options pure nomem nostack)." .
  requires ge a 0 .
do
  return asm_add2 k a b .
end
```

- 이 op 을 부르는 쪽은 **아무것도 적지 않는다.** 그것이 흡수의 값이다.
- `absorbs machine <이름>` 은 그 이름을 몸 안의 `cap machine` 으로 만든다. 그 권한이 태어나는 자리가 여기다 — 시작점이 받는 권한 목록에 `machine` 은 없다.
- 흡수할 수 있는 것은 `machine` 하나뿐이다. 입출력·씨·무더기처럼 **세상에 닿는** 권한은 흡수하지 못한다(`E-ABSORB-SCOPE`). 그런 권한을 만들어 내면 부르는 쪽이 못 본 권위가 생긴다.

부름 사슬로 그리면 차이가 이렇다.

```
흡수 없이                                  흡수와 함께
main      effects unsafe  ▲                main      (적을 것 없음)
add2      effects unsafe  | 위로 번진다    add2      absorbs machine k ◀ 여기서 멈춘다
asm_add2  effects unsafe  |                asm_add2  effects unsafe
```

갖추지 않으면 거절된다. 효과 줄이 `none` 이 아니면 `E-ABSORB-IMPURE`, 참조 구현이 없으면 `E-ABSORB-NOREF`, 계약(`requires`)이 없으면 `E-ABSORB-NOCONTRACT`, 근거(`why`)가 비었으면 `E-ABSORB-NOWHY` 다.

누가 흡수해도 되는가는 매니페스트가 정한다. 흡수는 «사람이 보증한다» 는 말이므로, 그 권리를 원하는 소스가 스스로에게 줄 수 없다. `pkg.low` 에 `build absorb <모듈> .` 로 적힌 모듈만 쓸 수 있고, 적히지 않으면 `E-ABSORB-PLACE` 다. 매니페스트가 아예 없으면 아무도 흡수하지 못한다.

도구가 그 자리를 말한다. `lowentc --absorbs f.low` 는 어느 op 이 무엇을 흡수했고 근거가 무엇인지 한 줄씩 낸다. 못 보는 구역은 숨기는 것이 아니라 보이게 둔다.

```
  add2                    absorbs machine as `k`  ref=add2_soft  line 40
      why: 레지스터 둘을 더할 뿐 메모리를 만지지 않는다…
absorbs: 1 op(s) stop `unsafe` here
```

타이밍은 적어 두는 것이다. 값에 따라 시간이 달라지는지는 도구가 확인하지 못한다. 그래서 흡수 장부의 타이밍 칸에 **사람이 적고**, 비워 두면 검사가 통과시키지 않는다 — «모른다» 도 답이다.

> **왜 참조 구현을 요구하나**
>
> 같은 답을 내는 순수한 판이 없으면 «옳음» 과 «일관되게 틀림» 을 가를 길이 없다. 실제로 이 저장소에서 그 일이 있었다: 기계 명령으로 다시 쓴 GHASH 가 **봉인하고 다시 여는** 시험을 통과했는데, 곱셈이 통째로 다른 연산이었다. 봉인과 개봉이 같은 코드를 쓰므로 스스로 일관되기만 하면 왕복은 성립한다. 잡은 것은 셈 판과의 대조였다.

30.7 기계가 가진 것을 쓸까 — `--hw`

기계 명령을 **쓸지**는 짓는 사람이 정한다.

고르는 것	무엇이 일어나나
`--hw none`(기본)	모든 셈을 평범한 코드로. 어느 기계에서나 선다

`--hw pclmul,aes,sse2,avx2`	그 명령이 **있다고 보고** 낸다. 없는 기계에서는 돌지 않는다
`--hw auto`	전부 담고 **시작할 때 한 번** 골라 고정한다. 어디서나 서고, 있는 기계에서는 빠르다. VAES·VPCLMULQDQ(ymm 한 칸에 두 블록)까지 담는다 — 처음 쓸 때 AES-NI 판과 맞대 보고 다르면 쓰지 않는다

표 30.3 — `--hw` 가 고르는 것

담는 것은 **셈을 대신하는 명령**만이 아니다. `sse2` · `avx2` 가 주는 것은 **폭**이다 — 서로 독립인 일 넷·여덟을 한 레지스터에 나란히 싣는 자리이고, ChaCha20 의 블록들이 바로 그런 모양이다. 규칙은 같다: 답은 같고, 없는 기계에는 담지 못하며, VM 은 언제나 평범한 코드로 돈다.

- 어느 쪽을 고르든 **답은 같다.** 다른 것은 빠르기와 타이밍 성질뿐이다 — 표를 읽는 셈은 읽는 자리가 값에 달렸고, 이 명령들은 그렇지 않다.
- 기계에 없는 명령을 담으라고 적으면 번역이 거절된다(`E-HW-TARGET`). 조용히 평범한 코드로 바꾸지 않는다 — 무엇이 돌지 짓는 사람이 알아야 한다.
- VM 은 언제나 평범한 코드로 돈다. 그래서 «VM 과 네이티브가 같은 답인가» 라는 이 저장소의 오라클이 곧 기계 경로의 시험이 된다.

30.8 흔한 실수

반례. 쓰기 전용 레지스터를 읽어 방금 쓴 값을 확인한다

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 ✘ 방금 무엇을 썼는지 확인하려고 쓰기 전용 레지스터를 읽는다
  return read_volatile g bsrr .
end
```

실행 결과

```
$ 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
```

보통 변수라면 쓰고 읽어 확인하는 것이 좋은 습관이지만, 쓰기 전용 레지스터를 읽으면 쓰레기가 나오거나 읽는 행위가 장치를 움직인다. 그래서 `E-MMIO-PERM` 이다. 방금 쓴 값이 필요하면 쓰기 전에 지역 변수에 남겨 두고, 장치의 실제 상태는 데이터시트가 정한 읽기 레지스터 (여기서는 `idr`)로 확인한다.

반례. 인터럽트 처리기에 매개변수를 둔다

examples/ch30/mistake_isrparams.low

```
module mistake_isrparams .
rem flags: --target cortex_m
rem expect: E-ISR-PARAMS

rem ✘ 몇 번 핀이 울렸는지 인자로 받으려 한다 --- 기계는 인자를 건네지 않는다
proc on_exti
  vector 6 .
  input pin u32 .
  output void .
  effects device .
do
  return .
end
```

실행 결과

```
$ 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
```

기계는 처리기를 부를 때 인자를 건네지 않는다. 어느 핀이 울렸는지는 처리기 안에서 장치의 상태 레지스터를 읽어 안다. 보통 코드와 나눌 값은 우선순위와 큐의 규율로 오간다(`spsc` 링 버퍼). 그래서 `E-ISR-PARAMS` 다.

반례. 인터럽트 처리기의 효과 줄을 비운다

examples/ch30/mistake_isreffect.low

```
module mistake_isreffect .
rem flags: --target cortex_m
rem expect: E-ISR-EFFECT

rem ✘ 할 일이 아직 없어 효과 줄을 비웠다
proc on_exti
  vector 6 .
  output void .
do
  return .
end
```

실행 결과

```
$ 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
```

할 일이 아직 없어도 인터럽트 처리기는 장치 때문에 있는 자리다. `effects device .` 를 적어야 이 op이 장치 쪽 코드라는 것이 머리에 남고, 등급(`build tier`)과 권한 검사가 그 사실을 따라간다. 적지 않으면 `E-ISR-EFFECT` 다.

반례. `asm` 의 기계 이름을 다른 도구의 철자로 적는다

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 ✘ 기계 이름을 다른 도구의 철자(`aarch64`)로 적었다 --- 이 처리기의 이름은 `arm64`
다
  asm aarch64 .
    reg a .
    reg b .
    out reg r .
do
  text ASM
    add {r}, {a}, {b}
  ASM
end
```

실행 결과

```
$ 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 나 LLVM 은 `aarch64` 라고 부르지만 이 처리기의 대상 이름은 `arm64` 다. 이름은 닫힌 목록(`x86_64`·`arm64`·`cortex_m`·`riscv64`·`mips_be`) 이고, 목록 밖의 이름은 어떤 빌드와도 맞지 않으므로 그 op 은 영영 지어지지 않는다. 그래서 조용히 건너뛰지 않고 `E-ASM-TARGET-UNKNOWN` 으로 멈춘다.

반례. 레지스터를 보통 칸처럼 읽고 쓴다

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 ✘ 레지스터를 보통 칸처럼 쓰고 읽는다 --- 합치거나 지워도 되는 접근으로 읽힌다
  set (field g moder) 2 .
  return field g idr .
end
```

실행 결과

```
$ 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` 와 `field g idr` 는 거절된다(`E-MMIO-PLAIN`). 보통 칸 접근은 처리기가 합치거나 지워도 되는 연산이기 때문이다 — 같은 레지스터를 두 번 읽는 코드가 한 번으로 줄거나, 아무도 읽지 않는 쓰기가 사라져도 보통 메모리에서는 답이 같다. 장치에서는 그 접근 자체가 일이므로 답이 달라진다. 장치 레지스터에는 `read_volatile`·`write_volatile` 을 쓴다 — 닿는 법(`ro`·`wo`)도 그 자리에서 함께 검사한다.

30.9 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`struct gpio do mmio 0x40020000 . moder u32 rw . … end`	장치의 레지스터 지도	새 낱말 없이 구조체에 절 하나
`rw` • `ro` • `wo`	닿는 법 — 번역에서 강제	어기면 `E-MMIO-PERM`
`read_volatile g idr` • `write_volatile g moder 2`	합치거나 지우지 않는 접근	읽는 행위 자체가 일이다
`input dev cap mmio .` + `effects device`	장치 권한과 효과	권한 없는 하드웨어 접근이 없다
`var g gpio be view gpio regs .`	바이트 위에 지도를 얹는다	장치를 값으로 받으면 `E-MMIO-BYVALUE`
`proc on_exti vector 6 . priority 2 . output void . effects device .`	인터럽트 처리기	부르면 `E-ISR-CALLED` · 인자는 `E-ISR-PARAMS`
`build tier t1 .`	이 기계가 감당하는 효과의 등급	실을 수 없는 것을 번역에서 막는다
`asm x86_64 . reg a . out reg r . clobber flags . options pure .`	기계 명령으로 쓴 몸의 머리	`unsafe`·`cap machine`·효과 줄·기계 이름으로 가둔다
`text ASM … {a} … ASM`	템플릿 — 피연산자와 맞대어 검사	`E-ASM-UNBOUND` · `E-ASM-UNUSED` · `E-ASM-OPTLIE`

표 30.4 — 하드웨어의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`mmio <주소>` 를 붙인 구조체가 장치의 지도이고, `read_volatile`·`write_volatile` 이 합치거나 지우지 않는 접근이다. `cap mmio` 와 `device` 효과가 필요하며, `ro`·`wo` 표지는 번역에서 강제되고 장치는 값으로 받을 수

> 없다. `vector` 절의 op 은 인터럽트 처리기로, 부를 수도 인자를 받을 수도 없다. `build tier` 는 기계가 감당하는 효과를 정한다. `asm` 절은 기계 명령을 `unsafe· cap machine·효과 줄·기계 이름`으로 가두고, 피연산자와 템플릿을 양쪽으로 맞댄다.

31 짓고 시험하기 — 패키지, 구성, 시험, 대조

이 장의 차례

31.1 매니페스트와 프로젝트 383
31.2 의존은 해시로 고정한다 383
31.3 빌드마다 다른 프로그램 — `build option` 과 `config` 384
31.4 시험 386
31.5 컴파일러를 검증하는 법 388
31.6 느린 자리를 묻는다 389
31.7 흔한 실수 389
31.8 이 장의 문법 한눈에 392

먼저 알아야 할 것

2장 첫 프로그램 · `lowentc` 의 모드와 두 백엔드
21장 모듈 · 검색 경로가 없다
26장 태스크와 채널 · `schedule explore_interleavings` 시험

돌아보기

2장에서 VM 과 네이티브가 다른 출력을 내면 누구의 결함이라고 했는가? 그리고 `--check` 가 조용하다는 것은 무엇을 보증하지 **않는가**?

> **답.** 컴파일러의 결함이라고 했다. 두 백엔드는 같은 중간 표현에서 출발하기 때문이다. `--check` 가 조용하다는 것은 머리의 약속과 본문이 어긋나지 않는다는 데까지이고, 시험이 통과했다거나 프로그램이 옳다는 것을 보증하지 않는다. 이 장은 그 둘 사이를 채우는 도구들 — 시험, 구성, 대조 — 과 프로젝트를 짓는 법을 다룬다.

이 장의 필요성과 맥락

제8부의 마지막 장이다. 앞의 장들은 파일 하나를 `--check` 하고 `--run` 했다. 실제 프로젝트는 매니페스트가 있고, 빌드마다 다른 구성이 있고, 시험이 있고, 의존이 있다. Lowent 의 도구는 이 모두에 같은 원칙을 적용한다 — 적은 것은 검사하고, 기본값은 소스에 보이며, 조용히 달라지는 것은 없다. 그리고 컴파일러 자신도 같은 원칙으로 검증된다.

이 장이 끝나면

`pkg.low` 매니페스트와 `lowentc run·build` 로 프로젝트를 짓는 법을 익힌다. `build option` 과 `config` 로 빌드마다 다른 프로그램을 짓되 꺼진 가지도 검사받는다는 것을 알게 된다. `test` 블록이 통과할 때와 실패할 때의 모습, 두 백엔드 대조와 계약 대조가 컴파일러를 어떻게 검증하는지, 의존을 내용 해시로 고정하는 방법도 보게 된다.

이 장에서 답할 질문

1 시험이 있는데 계약까지 적어야 하는가?

31.1 매니페스트와 프로젝트

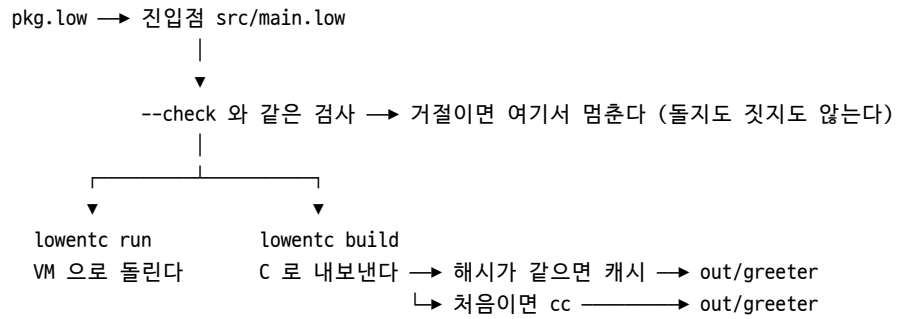
프로젝트의 뿌리에 `pkg.low` 를 두고, 진입점은 `src/main.low` 에 둔다.

```
package name "greeter" .
package version "0.1.0" .
package license "MIT" .
```

매니페스트는 평평한 선언 몇 줄이다. 열쇠말은 닫힌 집합이고 `version` 은 판 번호 규칙(semver)으로 검사된다. 도구와 사람이 소스 전체를 읽지 않고 이것 하나로 프로젝트의 정체를 알게 하려는 파일이라서, 여기만은 적은 것을 검사한다. 읽어서 믿을 수 없다면 없는 것과 같다.

```
$ lowentc run
hello from a package
main() = 0
$ lowentc build
built …/out/greeter
$ ./out/greeter main
hello from a package
```

`lowentc run` 은 매니페스트를 찾아 진입점을 VM 으로 돌리고, `lowentc build` 는 C 로 내보내 `out/` 에 실행 파일을 짓는다. 빌드는 방출된 C 의 내용 해시로 캐시되어, 바뀌지 않은 코드는 다시 컴파일하지 않는다. 매니페스트는 첫 파일의 폴더에서 위로 걸어 올라가며 찾는다.



두 명령 모두 짓거나 돌리기 **전에** `--check` 와 같은 검사를 거친다. 언어가 거절한 프로그램은 돌지도 않고 실행 파일이 되지도 않는다. `run` 에는 2026-09-16 부터, `build` 에는 2026-09-26 부터 그렇다 — 그 전의 `build` 는 검사를 건너뛰고 `return true .` 같은 잘못이 든 프로그램도 지었다(이 장을 다시 쓰며 실측해 찾았다).

31.2 의존은 해시로 고정한다

`lowentc add <이름> <자리>` 는 의존을 매니페스트에 **해시 핀과 함께** 적는다. `--lock-write` 는 지금 무엇을 상대로 빌드하는지를 락 파일로 찍고, `--lock <파일>` 은 그 해시와 바이트가 다르면 빌드를 거절한다. 판 번호가 같아도 바이트가 바뀌었으면 다른 의존이다.

진본인지를 확인하는 층은 따로 있다. `lowentc key new·sign·verify` 가 ed25519 분리 서명을 만들고 확인한다. `verify` 는 답을 네 층으로 나누어 말한다 — 무결(해시와 핀이 맞는가), 진본(신뢰하는 키로 서명되었는가), 전송, 접근. 한 층의 초록이 다른 층의 초록을 뜻하지 않기 때문이다. 설정을 찾으려고 읽는 환경 변수는 `HOME` 하나이고, 설정의 우선순위(명령 줄 > 프로젝트 > 사용자 > 전역)는 도움말에 적혀 있다. 소스 밖에서 동작을 몰래 바꾸는 자리를 줄이려는 것이다. 그 밖에 이름이 `LOW` 로 시작하는 스위치가 몇 있으나 모두 기본은 꺼져 있고, 시험과 옮겨 가기를 위한 것이다 — 결함 주입기 `LOW_HOST_FAULT`(28장), 없앤 붙임점 필드 접근을 잠시 되받는 되돌림 문 `LOWENT_ALLOW_GLUED_FIELD`, 대조 도구의 걸음 상한 `LOWENT_ORACLE_BUDGET` 같은 것이다. 평소에 켤 일이 없고, 켜면 그 사실이 명령 줄에 보인다.

`verify` 가 네 층을 따로 말하는 모습은 이렇다(서명하지 않은 파일에서 실측).

```
[3 integrity]     BLAKE3 a19e7c93…      이 바이트가 핀과 같은가
[2 authenticity]  unsigned — LOCAL       누가 냈는가 --- 믿는 키의 서명
[1 transport]     TLS (curl 의 몫)       서버가 누구인가 --- 바이트가 아니라
[4 access]        도구 설정의 토큰       들어갈 수 있는가 --- 무결과 별개
```

한 층이 초록이어도 다른 층은 따로 본다. TLS 가 성공했다는 것은 서버가 맞다는 뜻이지 받은 바이트가 맞다는 뜻이 아니다.

31.3 빌드마다 다른 프로그램 — `build option` 과 `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
```

실행 결과

```
$ lowentc --run tick_rate knobs.low
tick_rate() = 250
$ lowentc --run cpus knobs.low
cpus() = 64
```

`build option <이름> <갈래> …` 이 손잡이를 선언한다. 갈래는 `bool`·`int`·`choice` 셋이다. `config <이름>` 은 그 값을 **번역 시점 상수**로 읽는다. 구성 파일을 주지 않으면 선언의 기본값이 쓰인다.

같은 소스를 다른 구성으로 짓는다. 구성 파일은 `이름 값` 줄의 모음이다.

```
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
```

실행 결과

```
$ lowentc --config small.config --run tick_rate knobs_small.low
tick_rate() = 100
$ lowentc --config small.config --run cpus knobs_small.low
cpus() = 8
```

`smp` 를 끄자 `tick_rate` 는 100 을, `cpus` 는 8 을 낸다. 값이 정해지면 꺼진 가지는 생성물에 남지 않는다. 비용이 0 이다.

```
build option smp bool default true .       ← 소스가 손잡이를 선언한다
small.config:  smp false                   ← 구성 파일이 값을 준다 (없으면 기본값)

if config smp . do return config hz . end  ← smp 가 켜졌을 때의 가지
return 100 .                               ← 꺼졌을 때의 가지

번역   두 가지 모두 파싱하고 타입 검사한다
생성물 고른 가지만 남는다 (꺼진 가지의 비용 0)
```

C 의 `#ifdef` 와 결정적으로 다른 점이 있다. **꺼진 가지도 파싱되고 타입 검사를 받는다.** C 에서는 켜지지 않은 조합의 코드가 아무에게도 읽히지 않은 채 썩어서, 커널 규모의 프로젝트에서 "그 옵션 조합은 빌드조차 안 된다" 는 일이 생긴다. 여기서는 접히는 것이 **코드 내기**이지 검사가 아니다.

선언해 놓고 아무 코드도 읽지 않는 손잡이는 거절된다.

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
```

실행 결과

```
$ 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
```

그런 손잡이는 구성에 나타나고, 쓰는 사람이 그것을 끄는데, 아무 일도 일어나지 않는다. 결정처럼 보이지만 결정이 아니다. 구성 파일이 선택지에 없는 값을 주거나(E-CONFIG-TYPE), 없는 손잡이를 적어도(E-CONFIG-UNDEF) 거절된다.

> **실제 사례. 끌 수 없는 손잡이**
>
> 명세의 예는 hz 손잡이가 smp 에 기대게(depends smp) 적는다. 기대는 쪽이 켜져 있는데 기댐을 받는 쪽이 꺼져 있으면 그런 빌드는 없으므로 거절한다는 규칙이다. 이 책을 쓰며 그 예를 돌려 보니, choice 손잡이는 늘 값이 있어 "켜짐" 으로 읽혀서 smp 를 끄는 구성이 모두 E-CONFIG-DEPENDS 로 거절되었다. 개발 저장소의 시험은 그 구성으로 실행만 보고 검사는 보지 않아 이 어긋남을 놓쳤다. 이 장의 예제가 depends 를 뺀 이유다. 시험이 보는 것과 사용자가 하는 것이 다르면 초록불이 아무것도 말하지 않는다는 교훈이 여기에도 있다.

31.4 시험

test <이름> do … end 블록 안의 expect 가 단언이다. --test 가 모든 시험을 돌린다.

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
```

실행 결과

```
$ lowentc --test tests_clause.low
  [PASS] clamp_keeps_small
  [PASS] clamp_saturates
== tests: 2 run, 2 passed, 0 FAILED ==
```

시험이 실패하면 이렇게 나온다.

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
```

실행 결과

```
$ 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` 은 계약 위반과 다른 진단이다. 계약 위반은 **코드가 자기 약속을 어긴 것**이고, 시험 실패는 **시험이 코드가 틀렸다고 말하는 것**이다. 고칠 것이 다르다. 여기서는 `narrow_wrap` 을 `narrow_sat` 로 바꿔야 한다. 그리고 처리기는 `expect` 를 최적화로 없애지 않는다. 사라진 시험은 돌지 않은 시험이다.

동시성 코드는 `test <이름> schedule explore_interleavings . do … end` 로 가능한 모든 흐름의 차례를 돌려 답이 같은지 본다 (26장). 경우가 많으면 `limit <수>` 로 상한을 둔다.

op 머리에도 문서와 시험을 적는 절이 있다.

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 — 사람이 읽는 설명. 도구가 문서로 뽑는다
  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 — 이 op 을 시험하는 op 의 이름. 도구는 그 이름이 실제로 있는지 검사한다
  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 test 블록이 시험 op 들을 부른다 — lowentc --test 가 돌린다
test first_two_cases do
  expect first_two_ok .
  expect first_two_short .
end
```

실행 결과

```
$ 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 "…" .` 은 사람이 읽는 설명이다. 주석(`rem`)과 달리 op 에 딸린 문서로 남아 도구가 뽑아 쓴다. 길면 `lowdoc text DOC … DOC .` 로 여러 줄을 적는다(3장).
- `tests first_two_ok first_two_short .` 는 이 op 을 시험하는 **op 의 이름**이다. `test` 블록 이름이 아니다. 도구는 그 이름이 실제로 있는지 검사하고, 없으면 `E-CONTRACT-UNDEF` 로 거절한다. 시험이 이름을 바꾸거나 사라지면 머리가 그 사실을 알려 준다.
- `errors too_short lt (len data) 2 .` 는 조건이 붙은 실패다. 시험 op 둘이 성공과 실패 갈래를 하나씩 확인하고, `test` 블록이 둘을 `expect` 로 묶는다.

> **문. 시험이 있는데 계약까지 적어야 하는가?**
>
> **답.** 둘은 다른 것을 잡는다. 시험은 저자가 고른 입력에서 답이 맞는지 보고, 계약은 모든 호출에서 약속이 지켜지는지 본다. 그리고 계약은 시험을 **만드는 재료**가 된다. 개발 저장소의 계약 대조 도구는 기대 출력을 따로 적지 않고 `requires·ensures·errors` 를 판정 기준으로 삼아 경계값 입력을 만든다. 계약을 정직하게 적으면 시험이 공짜로 늘어난다.

31.5 컴파일러를 검증하는 법

증명(제10부)은 **모델**에 대한 것이다. 실제 컴파일러가 그 모델을 따라가는지는 따로 확인해야 한다. 원리는 하나다 — **같은 것을 두 가지 방법으로 만들고, 답이 다르면 결함으로 본다.**

방법	무엇과 무엇을 맞대나	무엇을 잡나
두 백엔드 대조	VM 실행 ↔ 네이티브 실행	두 백엔드가 다른 답을 내는 곳
계약 대조	적어 둔 계약 ↔ 실제 실행 결과	계약이 사실과 다른 곳
분석 자기 점검	"안전하다" 고 믿은 색인 ↔ 실제 색인	분석이 틀린 곳(`E-VM-ANALYSIS`)
증명서 검산	검사를 지운 근거 ↔ 독립 검산기의 산술	규칙을 잘못 적용한 곳
한 줄 바꾸기	프로그램 ↔ 사실 하나를 지운 프로그램	정답을 몰라도 관계가 깨진 곳

표 31.1 — 컴파일러를 맞대는 방법

이런 검사는 결함이 **있음**을 보이지만 **없음**을 보이지는 못한다. 두 구현이 같은 답을 낸다고 둘 다 맞는 것은 아니다. 없음은 증명의 몫이다. 공개 저장소에서 직접 돌려 볼 수 있는 것은 `impl/` 의 `make check` 다. 단위 시험, 표준 라이브러리 전체 검사, 여러 op 의 VM·네이티브 대조, 거절되어야 하는 프로그램의 진단 코드를 한 번에 본다. 이 책의 검증 스크립트도 같은 대조를 모든 예제에 적용한다 — 그리고 그 대조가 이 책을 쓰는 동안 컴파일러의 어긋남 여럿을 드러냈다.

> **흔한 오해. 초록불이 켜지면 검사한 것이다**
>
> 초록불은 **본 것**만 말한다. 대조 목록에 없는 연산은 초록으로도 빨간불로도 보이지 않는다. 그래서 개발 저장소는 대조가 한 번도 보지 못한 옵코드를 **세어서** 말하게 한다. 위의 "끌 수 없는 손잡이" 도 같다. 시험이 실행만 보고 검사를 보지 않았으므로 초록이었다. 세어지지 않는 것은 관리되지 않는다.

31.6 느린 자리를 묻는다

네이티브 방출은 op 을 두 길로 내린다. 타입이 확정된 op 은 C 의 정수와 배열로 자연스럽게 내려가고, 그렇지 못한 op 은 태그를 단 값으로 도는 느린 길에 남는다. `--why-slow` 는 느린 길에 남은 op 과 그 이유를 말한다.

```
$ lowentc --why-slow bounds.low
why-slow: 0 / 1 op(s) still on the tagged path
```

성능이 서명에 보이지 않는 모델은 그 자체로 소스 밖의 지식이 된다. 그래서 도구가 말한다. `--no-fast` 는 모든 op 을 느린 길로 내리는 대조 스위치다. 두 길이 같은 답을 내는지 볼 때 쓴다.

31.7 흔한 실수

> **반례. 구성 파일에 손잡이 이름을 잘못 적는다**

`examples/ch31/mistake_configtypo.low`

```
module mistake_configtypo .
rem flags: --config typo.config
rem expect: E-CONFIG-UNDEF

rem ✘ 구성 파일에 `maxcpu` 를 `maxcpus` 로 적었다

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
```

실행 결과

```
$ 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` 은 아무도 읽지 않는 설정이다. 조용히 넘기면 `maxcpu` 는 기본값 64 로 남고, 쓰는 사람은 8 로 지었다고 믿는다. 그래서 `E-CONFIG-UNDEF` 로 멈춘다. 진단의 줄 번호가 `0:0` 인 것은 잘못이 소스가 아니라 구성 파일에 있기 때문이다.

반례. `choice` 손잡이에 선택지에 없는 값을 준다

examples/ch31/mistake_configvalue.low

```
module mistake_configvalue .
rem flags: --config badvalue.config
rem expect: E-CONFIG-TYPE

rem ✘ 구성 파일이 `hz` 에 선택지에 없는 300 을 준다

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
```

실행 결과

```
$ 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` 는 100 · 250 · 1000 가운데 하나다. 300 을 주면 가장 가까운 값으로 맞추거나 기본값으로 돌아가지 않고 `E-CONFIG-TYPE` 으로 멈춘다. 선택지는 그 값들로 시험되었다는 뜻이기도 하다. 새 값이 필요하면 소스의 선언에 선택지를 더한다.

흔한 오해. 권한을 받는 op 은 `--run` 으로 못 부른다

examples/ch31/runcap.low

```
module runcap .
rem run: say 5
rem run: say 0 5
```

```
rem `--run` 은 권한 자리를 **비워 두면 채워 준다**(2026-09-20). 자리표 `0` 을 넣던 옛 모
양도 그대로 돈다
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
```

실행 결과

```
$ lowentc --run say runcap.low 5
hi
say(5) = 3
$ lowentc --run say runcap.low 0 5
hi
say(0, 5) = 3
```

2026-09-20 까지는 그랬다: 권한 자리도 인자 수에 들어서 `say 5` 는 `E-VM-ARITY` 였고, 자리표 `0` 을 채운 `say 0 5` 만 돌았다. 그래서 `cap io` 나 `cap allocator` 를 받는 **증인 프로그램** 들은 네이티브로만 돌았고, 두 뒤끝을 맞대는 이 책의 규율이 그 자리에서 끊겨 있었다. 이제 도구가 권한 자리를 **채운다** — 인자를 덜 준 경우에만. 그래서 `say 5` 도 `say 0 5` 도 돈다. 자리표는 여전히 도구의 편의일 뿐이다: 프로그램 안에서 권한 자리에 수를 넘기면 `E-CAP-FORGE` 로 거절된다(16장).

흔한 오해. 구성에서 꺼진 가지는 검사하지 않는다

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 이 구성에서는 꺼진 가지다 --- 그래도 검사받는다
    return true .
  end
  return 0 .
end
```

실행 결과

```
$ 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`
```

`smp` 를 끈 구성에서는 `return true .` 가 결코 돌지 않는다. 그래도 `u64` 자리에 `bool` 을 돌려주는 이 줄은 `E-TYPE-RETURN` 으로 거절된다. C 의 `#ifdef` 안 코드는 켜지지 않은 조합에서 아무에게도 읽히지 않지만, `config` 의 꺼진 가지는 파싱되고 타입 검사를 받는다. 값이 정해진 뒤 생성물에서 빠질 뿐이다. 그래서 어떤 구성으로 지어도 나머지 조합이 썩지 않는다.

31.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`package name "greeter" .` · `package version "0.1.0" .`	`pkg.low` 매니페스트	여기만은 적은 것을 검사한다 — 열쇠말은 닫힌 집합
`lowentc run` · `lowentc build`	프로젝트를 VM 으로 돌린다 · `out/` 에 짓는다	매니페스트를 걸어 올라가며 찾는다
`lowentc add <이름> <자리>` · `--lock-write` · `--lock`	의존을 해시와 함께 고정한다	판 번호가 같아도 바이트가 다르면 다른 의존
`build option smp bool default true .`	빌드 손잡이 — `bool`·`int`·`choice`	읽히지 않는 손잡이는 `E-OPT-UNUSED`
`config smp` · `--config small.config`	손잡이 값을 번역 시점 상수로 읽는다 · 구성 파일	꺼진 가지도 검사받는다
`test <이름> do expect <조건> . end` · `--test`	시험 블록과 단언	실패는 `E-TEST-FAIL` — 계약 위반과 다른 진단
`test … schedule explore_interleavings limit <수> . do … end`	모든 차례를 돌리는 시험	드문 차례의 결함
`lowentc --run <op> <파일> <인자…>`	op 하나를 VM 으로 돌린다	권한 자리는 도구가 채운다 — 거절된 단위는 돌지 않는다
`--why-slow` · `--no-fast`	느린 길에 남은 op 과 이유 · 모두 느린 길로	성능을 도구가 말한다
`lowdoc "…" .` · `tests op1 op2 .`	op 에 딸린 문서 · 이 op 을 시험하는 op 이름	문서는 op 과 함께 움직이고, 없어진 시험은 머리가 알린다

표 31.2 — 빌드와 시험의 문법 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`pkg.low` 는 검사되는 매니페스트이고 `lowentc run`·`build` 가 프로젝트를 돌리고 짓는다. 의존은 내용 해시로 고정하고 진본은 서명으로 따로 확인한다. `build option` 과 `config` 는 빌드마다 다른 프로그램을 짓되 꺼진 가지도 검사받으며, 읽히지 않는 손잡이는 거절된다. `test` 블록의 실패는 계약 위반과 다른 진단이다. 컴파일러는 두 백엔드·계약·분석·증명서·한 줄 바꾸기로 맞대어 검증되고, `--why-slow` 는 느린 자리를 말한다.

제9부 — 표준 라이브러리

이 부는 표준 라이브러리를 둘러본다. 모듈마다의 자세한 설명은 부록 E 에 한 쪽씩 있고, 이 부는 그것을 되풀이하지 않는다. 대신 **어느 모듈이 어떤 층에 있고, 모두가 어떤 규약을 따르며, 무엇을 할 때 어디를 보면 되는지**를 예제와 함께 정리한다.

지도 (32장) — 언어·잎·라이브러리의 세 층, 라이브러리가 들어오는 기준, 순수 계산·저장·호스트의 층, 그리고 모든 모듈이 따르는 네 가지 규약.

쓰임새별 둘러보기 (33–37장) — 글자와 부호화, 그릇과 정렬, 저장소와 핸들, 입출력·네트워크·시간· 난수·암호, 터미널. 장마다 대표 모듈 몇을 실제로 돌려 보고, 나머지는 한 줄씩 자리를 알려 준다.

이 부를 읽는 동안 되풀이해서 보게 되는 것이 있다. 라이브러리는 특권이 없어서 앞의 여덟 부에서 배운 규칙 — 계약(op 이 스스로 적는 약속), 효과, 권한, 소유 — 이 그대로 적용된다. 새 낱말은 하나도 늘지 않았다. 그리고 모듈 문서의 첫머리는 늘 **무엇을 하지 않는지**를 적는다.

32 표준 라이브러리의 지도

이 장의 차례

32.1 세 층 — 언어, 잎, 라이브러리 395
32.2 무엇이 들어오는가 395
32.3 층으로 본 지도 396
32.4 파일 이름과 모듈 이름 396
32.5 모든 모듈이 따르는 규약 397
32.6 잎 op 의 목록 — 모듈 아래에 무엇이 있나 398
32.7 흔한 실수 400
32.8 이 장의 문법 한눈에 401

먼저 알아야 할 것

21장 모듈 · 표준 라이브러리 이름은 `use <이름> .` 으로 들여온다
16장 권한 · 권한은 건네받는 것이지 주워 쓰는 것이 아니다
17장 실패를 설계하기 · 경계에서는 실패를 값으로 돌려준다

돌아보기

21장에서 `use` 가 찾는 것은 파일 이름인가, 모듈 이름인가? 표준 라이브러리에서 이 차이가 왜 걸린다고 했는가?

> **답.** 파일 안의 `module` 선언이다. 표준 라이브러리에는 파일 이름과 모듈 이름이 다른 곳이 몇 있어서(`lib/alloc.low` 는 `allocs`, `lib/str.low` 는 `strings`), 파일 이름으로 들여오려다 헤맨다. 이 장의 첫 표가 그 목록이다.

이 장의 필요성과 맥락

제9부는 표준 라이브러리를 둘러본다. 모듈마다 자세한 설명은 부록 E 에 한 쪽씩 있고, 이 부는 그것을 되풀이하지 않는다. 대신 **어떤 모듈이 어떤 층에 있고, 모두가 어떤 규약을 따르며, 무엇을 할 때 어디를 보면 되는지**를 정리한다. 그 첫 장인 이 장은 지도다 — 언어와 라이브러리의 경계, 라이브러리가 들어오는 기준, 모듈의 층과 공통 규약.

이 장이 끝나면

언어·잎(leaf)·라이브러리 세 층을 가르는 물음을 알게 된다. 표준 라이브러리가 들어오는 기준(헌장)과 성숙도 사다리, 모듈을 할당하지 않는 층·저장이 필요한 층·권한이 필요한 층으로 나눈 지도를 익힌다. 호출자의 버퍼, 값으로 돌아오는 실패, 전량 아니면 무, 갚아야 하는 소유라는 공통 규약과, 파일 이름과 모듈 이름이 다른 곳들도 보게 된다.

이 장에서 답할 질문

1 언어에 넣으면 편할 기능을 왜 라이브러리로 두는가?

32.1 세 층 — 언어, 잎, 라이브러리

층	누가 만드나	자격
언어	명세	라이브러리로 표현할 수 없거나, 표현하면 비용이 숨는다
잎(leaf)	처리기	운영체제·하드웨어에 닿는다. Lowent 로 쓸 수 없다
라이브러리	누구나	나머지 전부. Lowent 로 쓴다

표 32.1 — 세 층

어떤 조각이 잎인지를 가르는 물음은 하나다 — "이것을 Lowent 로 쓸 수 있는가." 쓸 수 있으면 라이브러리다. 파일을 열어 정수 핸들을 받는 것은 커널에 닿으므로 잎이다. 그 핸들을 잊을 수 없는 소유 값으로 감싸는 것은 Lowent 로 쓸 수 있으므로 라이브러리다. 파일을 통째로 읽어 주는 편의 op 도 라이브러리다. **편의는 잎의 자격이 되지 않는다.**

표준 라이브러리는 특권이 없다. 저자가 쓴 코드와 같은 규칙 — 계약, 효과, 권한 — 을 받는다. `strings.starts_with` 의 소스는 이렇다.

```
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
```

이 책에서 배운 것만으로 읽힌다. 라이브러리만 쓸 수 있는 숨은 통로가 있으면 그 통로를 지나는 순간 프로그램의 보장이 끊기기 때문이다.

> **문. 언어에 넣으면 편할 기능을 왜 라이브러리로 두는가?**
>
> **답.** 언어에 낱말이나 내장 연산을 더하면 모든 프로그램이 그것을 외워야 하고, 처리기가 그것을 특별하게 다뤄야 한다. 라이브러리로 두면 같은 규칙으로 검사되고, 쓰지 않는 프로그램에는 없는 것이다. 그래서 언어는 **표현할 수 없는 것**만 받아들인다. 표준 라이브러리의 할당기·그릇·파일 핸들이 모두 이 기준을 통과해 라이브러리에 있다. 편하다는 것은 언어에 들어갈 이유가 되지 않는다.

32.2 무엇이 들어오는가

라이브러리에 들어오는 것은 **있어서**가 아니라 **증거로** 들어온다. 명세의 헌장은 여섯 잣대를 둔다.

잣대	뜻
이 언어로 쓸 수 있으면 이 언어로 쓴다	잎은 바깥 권위를 쓰거나 이 언어가 표현하지 못할 때뿐이다
시그니처가 비용과 권한의 장부다	효과·권한·저장·할당·소유·완결이 경계에서 보여야 한다
셈과 권위를 가른다	꼴 짓기와 내보내기, 해석과 파일 열기, 난수의 다음 걸음과 엔트로피 얻기는 다른 것이다
자리는 여러 축으로 가른다	권위·저장·실행 등급·프로파일을 한 사다리로 눌러 담지 않는다
안에 있는 것은 작게 둔다	이식되는 바탕과 얇은 호스트 어댑터까지
모듈마다 증거가 있다	단위 시험·거절 시험·대조·실패 시험을 갖춘다

표 32.2 — 라이브러리 헌장

모듈은 저마다 **성숙도**를 적는다 — `experimental`(적혀 있을 뿐) · `incubating`(결정 문서가 있다) · `standard`(설명이 있고 회귀 시험이 이름으로 부른다) · `deprecated`(무엇으로 갈아탈지 적혀 있다). 사다리의 값은 위 칸

이 아니라 아래 칸에 있다. `experimental` 이라고 적는 것은 "아직 믿지 말라" 를 **소스에** 적는 일이다. 칸은 품질의 등수가 아니라 약속의 크기다.

명세는 라이브러리의 **목록**을 적지 않는다. 무엇이 있는지는 소스에서 매번 다시 세어 내는 쪽이 권위이고, 문서가 그것을 베끼면 베낀 순간부터 갈라진다. 실제로 있지도 않은 API 셋이 표에 있다는 이유로 사실로 읽힌 일이 있었다. 이 부의 표도 이 판의 소스를 보고 만든 것이며, 모듈별 자세한 쪽은 부록 E 에 있다.

32.3 층으로 본 지도

층	성격	대표 모듈
L0 순수 계산	`effects none` · 호출자의 버퍼	`strings strbuf fmt utf8 utf16 unicode codec regex hash math random sortlib sortgen searchlib hashmap strmap spsc term` · 암호 모듈들
L1 저장	할당 권한이나 빌린 바이트가 필요하다	`allocs pool shard budget segarena pagecache growvec vecgen mapgen nodelist segview wire flags lifemode`
L2 호스트	권한이 필요하다 (`cap io·file_system·tty·net·clock`)	`io outbuf files tty net clock`

표 32.3 — 표준 라이브러리의 층

이 부의 나머지 장은 쓰임새로 묶는다. 글자와 부호화(33장), 그릇과 정렬(34장), 저장소와 핸들 (35장), 입출력과 네트워크와 암호(36장), 터미널(37장)이다.

32.4 파일 이름과 모듈 이름

`use` 가 찾는 것은 파일 안의 `module` 선언이다. 몇 곳은 파일 이름과 다르다.

소스 파일	모듈 이름
`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`

표 32.4 — 파일 이름과 다른 모듈 이름

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
```

실행 결과

```
$ 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 .` 은 `from` 없이 표준 모듈 자리에서 풀린다. 들여온 이름은 언제나 `strings.starts_with` 처럼 모듈로 한정해 부른다. 글롭 들여오기는 없다. 파일 이름으로 들여오면 그런 모듈이 없다.

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
```

실행 결과

```
$ 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)
```

진단이 오류가 아니라 경고인 것은 검색 경로가 없어서 이 번역 단위만 보고는 그 이름이 어딘가에 있는지 없는지 알 수 없기 때문이다 (21장). 그 이름을 실제로 쓰는 순간 거절된다.

32.5 모든 모듈이 따르는 규약

규약	뜻
버퍼는 호출자의 것	L0 모듈은 할당하지 않는다. 출력 버퍼와 작업 공간을 매개변수로 받는다. 그래서 재진입이 공짜고, VM·네이티브 대조가 그대로 돈다
실패는 값	`option`(못 했다)이나 `result`(무엇이 실패했는지). 트랩도, 조용한 잘림도, 치환 문자도 없다
전량 아니면 무	자리가 모자라면 한 바이트도 쓰지 않는다. 절반 쓴 버퍼는 조용히 틀린 출력이다
소유는 갚아야 한다	완결이 필요한 것(`outbuf` 의 남은 바이트, `files` 의 핸들)은 소유 값이라 잊으면 번역이 거절한다

표 32.5 — 표준 라이브러리의 공통 규약

> **흔한 오해. 표준 라이브러리니까 모든 모듈을 똑같이 믿어도 된다**
>
> 모듈마다 성숙도가 다르고, 모듈 문서의 첫머리에 무엇을 **하지 않는지**가 적혀 있다. 예를 들어 `tlssrv` 는 핸드셰이크를 모두 하지만 전송은 아직이고, `der` 은 공개키만 꺼내는 최소 파서이지 인증서 체계(PKI)가 아니다. 들여오기 전에 그 모듈이 스스로 적은 경계를 읽는 것이 이 라이브러리를 쓰는 법이다.

32.6 잎 op 의 목록 — 모듈 아래에 무엇이 있나

라이브러리 모듈은 결국 처리기가 주는 **잎** 내장 op 을 부른다. 보통은 모듈을 쓰고 잎을 직접 부르지 않는다 — 모듈이 소유·세 자리 답·전량 아니면 무 같은 규약을 입혀 주기 때문이다. 그래도 어느 모듈이 무엇을 딛는지 알면 모듈 문서를 읽거나 새 모듈을 지을 때 길을 잃지 않는다.

잎 op	하는 일	감싸는 모듈 · 권한
`file_open` · `file_read` · `file_write` · `file_seek` · `file_close`	파일을 열고 읽고 쓰고 자리를 옮기고 닫는다	`files` · `cap file_system`
`dir_make` · `dir_read` · `dir_close`	디렉터리를 만들고 항목을 읽고 닫는다	`files` · `cap file_system`
`path_remove` · `path_rename` · `link_type`	경로를 지우고 이름을 바꾸고, 심링크를 따라가지 않고 그 자신이 무엇인지 묻는다	`files` · `cap file_system`
`net_listen` · `net_accept` · `net_connect` · `net_port`	듣는 소켓 · 연결 받기 · 주소와 포트로 연결 걸기 · 포트 묻기	`net` · `cap net`
`net_resolve`	이름을 IPv4 주소로 — DNS. 연결과 **가른 잎**이라, 주소를 이미 가진 프로그램은 이름 해석에 닿지 않는다	`net` · `cap net`
`net_send` · `net_recv` · `net_close` · `net_pair`	보내기 · 받기 · 닫기 · 맞물린 한 쌍	`net` · `cap net`
`reactor_new` · `r_read` · `r_write`	reactor 를 만든다 · reactor 로 읽는다 · reactor 로 쓴다	모듈 없음 — `cap io`
`env_get`	환경 변수를 읽는다	모듈 없음 — `cap env`
`rng_next`	상태에서 다음 난수(splitmix64)	`random` — 권한 없음(순수)
`hash_bytes` · `crc32`	FNV-1a 64 해시 · CRC-32 검사값	`hash` — 권한 없음
`sha256` · `sha384` · `sha512`	SHA-256 · SHA-384 · SHA-512 — SHA-384 은 SHA-512 의 다른 시작값이고 앞 48 바이트다	`hash·hmac·tls13` · `ed25519` — 권한 없음
`aes_ctr` · `ghash`	AES-128-CTR 흐름 · GHASH 누산 — 키·카운터·누산기가 모두 16 바이트다. 카운터와 누산기는 **제자리에서** 고쳐지므로 이어 불러도 이어진다	`gcm` — 권한 없음

`chacha_poly`	ChaCha20-Poly1305 한 덩이 — 흐름과 누산을 **한 바퀴에**. 뜻은 `chacha20` 을 돌리고 그 암호문을 `poly1305` 에 먹인 것과 **같다**. 두 셈이 쓰는 자리가 달라(벡터 · 정수 곱셈기) 섞으면 겹쳐 돈다(실측 1,080 → 1,199 MB/s)	`aead` — 권한 없음
`aes_gcm`	AES-128-GCM 한 덩이 — 흐름과 누산을 **한 바퀴에** 돈다. 뜻은 `aes_ctr` 를 돌리고 그 암호문을 `ghash` 에 먹인 것과 **같다**; 다른 것은 빠르기뿐이다(실측 2848 → 3661 MB/s). 봉인만 쓴다 — 복호는 태그를 먼저 증언하고 맞을 때만 풀어야 하므로 두 바퀴로 남는다	`gcm` — 권한 없음
`chacha20 · poly1305`	ChaCha20 흐름 · Poly1305 누산 — 키 32 · 카운터 블록 16(앞 4 가 블록 번호, 리틀엔디언)이고 카운터는 **제자리에서** 올라간다. `poly1305` 은 상태(h 다섯 · r 다섯)를 누산하고, 열여섯의 배수가 아니면 꼬리를 0 으로 채운다. 폭은 빌드가 정한다(`lowentc --hw sse2 · avx2`)	`chacha · poly · aead` — 권한 없음
`aes_round · aes_round_last`	AES 한 라운드 — 상태 16 바이트를 제자리에서. 기계에 그 명령이 있으면 처리기가 그것으로 내리고(`lowentc --hw aes`), 없으면 같은 답을 내는 표 셈으로 내린다	`aes` — 권한 없음
`str_from_cstr`	C 문자열(NUL 로 끝남)을 훑어 `str` 로 만든다	C 경계 — VM 은 `E-VM-CSTR` 로 못 한다고 말한다

표 32.6 — 잎 내장 op 과 그것을 감싸는 모듈

계산 잎은 `call_builtin` 뒤에서만 선다. `call_builtin sha256 msg out` 처럼 이름이 그 자리에만 살고 전역 어휘가 되지 않는다 — 한 번 쓰는 낱말로 모두가 외울 것을 늘리지 않으려는 규율이고, `pipe` 안의 단계 이름과 `cast u8 x` 의 타입 자리가 이미 같은 방식이다. 운영체제에 닿는 잎(`file_open` · `net_send` …)은 그대로 부른다 — 첫 피연산자가 권한이라 이미 특별함이 보인다.

잎의 자격은 "Lowent 로 쓸 수 있는가" 하나다. `rng_next` 와 해시 셋이 순수 계산인데도 잎인 것은, 알고리즘을 처리기가 고정해 VM 과 네이티브가 비트까지 같은 답을 내게 하려는 것이다. 권한이 필요한 잎은 모두 그 권한을 **첫 피연산자**로 받는다(16장).

32.7 흔한 실수

반례. `option` 으로 돌아오는 쓰기를 묻지 않고 이어 쓴다

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 ✘ 앞의 쓰기가 성공했다고 믿고 `some_value` 로 곧장 이어 쓴다
  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
```

실행 결과

```
$ 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)
```

`fmt` 의 쓰기 op 은 다음 쓸 자리를 `option u64` 로 준다. 열두 칸 버퍼에서는 셋 다 성공해 8 을 내지만, 여섯 칸에서는 `put_u64` 가 자리가 모자라 `none` 을 주고 `some_value` 가 멈춘다. L0 모듈의 실패는 트랩이 아니라 **값**이므로, 받는 쪽이 걸음마다 묻는다.

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 걸음마다 묻는다 --- 모자라면 0 을 돌려 "쓰지 못했다" 를 알린다
  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
```

실행 결과

```
$ 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]
```

고친 판은 여섯 칸 버퍼에서 0 을 돌려준다. 버퍼에 `hello ` 가 남아 있는 것에 주목한다. "전량 아니면 무" 는 **op 한 번**의 약속이지 여러 번 부른 묶음 전체의 약속이 아니다. 묶음 전체를 되돌려야 하면 성공한 자리(p·q)까지를 유효한 길이로 삼는다.

흔한 오해. 버퍼가 모자라면 들어가는 만큼은 쓴다

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 여섯 바이트를 네 칸에 --- 앞 네 바이트만 쓰는 대신 하나도 쓰지 않는다
  let p option u64 be fmt.put_str buf 0 "Lowent" .
  guard is_some p . else return 99 .
  return some_value p .
end
```

실행 결과

```
$ 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 의 `snprintf` 는 들어가는 만큼 쓰고 잘린다. 표준 라이브러리의 L0 모듈은 **전량 아니면 무**다. 여섯 바이트를 네 칸에 쓰려 하면 한 바이트도 쓰지 않고 `none` 을 주며, 결과의 버퍼는 `[0,0,0,0]` 그대로다. 절반 쓴 버퍼는 조용히 틀린 출력이 되고, 부르는 쪽은 "none 이면 버퍼는 그대로" 를 믿고 다른 길로 이어 쓸 수 있다.

32.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`use strings .`	표준 모듈을 들여온다 — 이름은 파일 안의 `module` 선언	`lib/str.low` 의 모듈은 `strings`
`strings.starts_with line "GET "`	들여온 이름은 모듈로 한정한다	글롭 들여오기가 없다
`fmt.put_str buf pos s → option u64`	호출자의 버퍼에 쓰고 다음 자리를 준다	L0 은 할당하지 않는다 — 재진입이 공짜
`none`**(자리가 모자람)**	한 바이트도 쓰지 않았다	전량 아니면 무 — op 한 번의 약속

`result t e`	무엇이 실패했는지 말하는 실패	트랩도 조용한 잘림도 없다
`owned` **핸들 · 남은 바이트**	갚아야 하는 소유	잊으면 번역이 거절한다
`experimental` · `incubating` · `standard` · `deprecated`	모듈이 스스로 적는 성숙도	칸은 등수가 아니라 약속의 크기

표 32.7 — 표준 라이브러리를 쓰는 모양 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

언어는 표현할 수 없는 것만, 잎은 바깥에 닿는 얇은 조각만 맡고, 나머지는 같은 규칙을 받는 라이브러리다. 라이브러리는 헌장의 증거로 들어오고 성숙도를 소스에 적는다. 모듈은 순수 계산(L0)·저장(L1)·호스트(L2) 층으로 나뉜다. 모듈 이름은 파일 이름과 다를 수 있다. 모든 모듈은 호출자의 버퍼·값으로 돌아오는 실패·전량 아니면 무·갚아야 하는 소유를 따른다.

33 글자와 부호화 — `strings`·`fmt`·`utf8`·`codec`·`hash`

이 장의 차례

33.1 자르고 찾고 조립한다 403
33.2 UTF-8 — 치환 문자를 만들지 않는다 405
33.3 16진과 base64 406
33.4 해시 — 세 가지 물음에 세 가지 답 406
33.5 정규식 — 역추적하지 않는다 407
33.6 흔한 실수 407
33.7 이 장의 문법 한눈에 410

먼저 알아야 할 것

9장 줄 · 문자열은 타입이 아니라 `slice u8` 이다
32장 표준 라이브러리의 지도 · 호출자의 버퍼·값으로 돌아오는 실패·전량 아니면 무

돌아보기

9장에서 "문자열은 따로 된 타입이 없는가" 라는 물음에 무엇이라 답했는가?

> **답.** 문자열 리터럴은 `slice u8` 이고, 글자가 몇 개인지나 UTF-8 로 올바른지는 타입이 아니라 라이브러리의 op 이 답한다고 했다. 바이트와 글자를 같은 것으로 여기는 순간 생기는 결함을 피하려는 선택이다. 이 장이 그 라이브러리를 둘러본다.

이 장의 필요성과 맥락

거의 모든 프로그램이 문자열을 자르고, 찾고, 이어 붙이고, 숫자를 글자로 바꾸고, 바이트를 16진이나 base64 로 옮긴다. C 에서는 이 모든 일에 널 종단 문자열과 넘치는 버퍼가 따라다닌다. Lowent 의 글자 모듈들은 할당하지 않고, 결과를 뷰나 호출자의 버퍼로 돌려주고, 실패를 값으로 알린다. 32장의 공통 규약이 가장 자주 보이는 자리라서 라이브러리 둘러보기의 첫 모듈 묶음으로 고른다.

이 장이 끝나면

`strings` 로 문자열을 뷰로 다루고, `fmt` 로 호출자의 버퍼에 글자를 조립하는 법을 익힌다. `utf8` 이 올바르지 않은 바이트를 치환 문자 없이 `none` 으로 알린다는 것, `codec` 의 16진·base64 와 `hash` 의 세 가지 해시(자리 고르기·손상 검출·SHA-256)가 각각 무엇에 쓰이는지 알게 된다. 그 밖의 글자 모듈(`strbuf`·`utf16`·`unicode`·`regex`)이 어디에 있는지도 보게 된다.

이 장에서 답할 질문

1 `fmt.put_str` 을 부를 때마다 `guard is_some` 을 적는 것이 번거롭지 않은가?

33.1 자르고 찾고 조립한다

`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
```

실행 결과

```
$ lowentc --run main request.low
path=/users/42 len=9
main() = 0
```

- `strings.starts_with`·`strings.remove_prefix`·`strings.find` 는 모두 `slice u8` 을 받는다. `remove_prefix` 와 `subslice` 가 돌려주는 것은 새 버퍼가 아니라 원래 바이트를 가리키는 **뷰**다. 복사가 없다.
- `fmt.put_str buf pos s` 와 `fmt.put_u64 buf pos n` 은 호출자의 버퍼 `buf` 의 `pos` 자리부터 쓰고, **다음 쓸 자리**를 `option` 으로 준다. 자리가 모자라면 한 글자도 쓰지 않고 `none` 이다.
- 조립이 끝나면 `subslice buf 0 끝` 을 표준출력에 쓴다.

버퍼를 `cap allocator` 로 받았으므로 머리에 `alloc` 이 보인다. `strings` 와 `fmt` 자체는 할당하지 않는다. 버퍼를 어디서 얻을지는 호출자의 일이다.

> **문. `fmt.put_str` 을 부를 때마다 `guard is_some` 을 적는 것이 번거롭지 않은가?**
>
> **답.** 번거롭다. 대신 한 번이라도 자리가 모자라면 그 자리에서 멈추고, 절반만 쓴 출력이 나가지 않는다. 버퍼를 넉넉히 잡고 반복을 줄이려면 `strbuf` 를 쓴다. `strbuf.append` 는 소유 버퍼에 이어 붙이고 실패를 `result` 로 준다. `strings` 가 읽기(뷰)라면 `strbuf` 는 쓰기다.

33.2 UTF-8 — 치환 문자를 만들지 않는다

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
```

실행 결과

```
$ 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` 는 글자 수를 센다. 한글 “안녕” 은 6 바이트, 2 글자다.
- 끝이 잘린 `[236,149]` 는 올바른 UTF-8 이 아니므로 `none` 이고, 예제는 999 를 돌려준다.
- `utf8.decode s 0` 은 0 번 자리의 글자 하나를 코드포인트로 준다. 50504 는 U+C548, “안” 이다.

많은 라이브러리는 올바르지 않은 바이트를 만나면 치환 문자(U+FFFD)를 넣고 계속 간다. 그러면 입력이 망가졌다는 사실이 출력 속에 묻힌다. `utf8` 은 `none` 으로 알린다. 망가진 입력을 어떻게 다룰지는 부르는 쪽이 정한다.

UTF-16 을 쓰는 세계(Windows API·Java·JavaScript)와 값을 주고받을 때는 `utf16` 이 `slice u16` 위에서 서로게이트 쌍을 합치고 쪼갠다. 짝이 맞지 않으면 역시 `none` 이다. 글자인지·숫자인지·공백인지·폭이 0 인지를 묻는 표는 `unicode` 모듈에 있다. 그 표는 유니코드 자료에서 기계적으로 뽑았다. 손으로 고른 판정은 빠뜨린 구간에서 오류 대신 조용히 `false` 를 내기 때문이다.

33.3 16진과 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
```

실행 결과

```
$ lowentc --run main hexout.low
4c6f77656e74
main() = 0
```

`codec.hex_enc src dst` 는 `src` 를 16진 글자로 `dst` 에 쓰고 **실제로 쓴 바이트 수**를 준다. 결과는 `dst` 통째가 아니라 앞의 `n` 바이트다. 여섯 글자 “Lowent” 는 열두 글자 `4c6f77656e74` 가 된다. 8 바이트 자리에 18 글자를 부호화하려 하면 `none` 이고 아무것도 쓰지 않는다 — 전량 아니면 무다. `hex_dec·b64_enc·b64_dec` 도 같은 모양이다.

> **흔한 오해. 부호화는 암호화의 한 가지다**
>
> 16진과 base64 는 바이트를 글자로 **옮겨 적는** 방법일 뿐 비밀을 지키지 않는다. 누구나 되돌린다. 비밀이 필요하면 봉인하는 `aead` 를 쓴다(36장). `codec` 과 `aead` 가 다른 모듈인 것은 헌장의 “셈과 권위를 가른다” 와 같은 까닭이다 — 다른 일에 한 이름을 주지 않는다.

33.4 해시 — 세 가지 물음에 세 가지 답

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
```

실행 결과

```
$ 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]
```

`hash` 모듈은 서로 다른 물음에 서로 다른 해시를 준다.

물음	op	쓰는 해시
해시맵의 어느 칸에 넣나	`of` · `bucket_of` · `bucket_mask`	FNV-1a — 빠르고 고르다
저장·전송 중에 손상됐나	`crc` · `crc_ok`	CRC-32 — 우연한 비트 오류를 잡는다
누가 일부러 바꾸지 않았나	`digest` · `digest_ok`	SHA-256 — 충돌을 찾기 어렵다

표 33.1 — `hash` 모듈이 답하는 물음

`slot` 은 키 “hello” 를 16 칸 가운데 11 번에 넣는다. `bucket_of` 는 나머지 `mod` 로 칸을 고르므로 결과가 언제나 칸 수보다 작고(4장), 그래서 해시 테이블의 색인 경계 검사가 지워진다. 칸 고르기용 해시로 변조를 막으려 하거나, 암호 해시로 해시맵 칸을 고르면 둘 다 틀린 도구를 쓴 것이다. 이름이 물음을 가른다.

33.5 정규식 — 역추적하지 않는다

`regex` 는 패턴을 한 번 컴파일해 프로그램(`slice u64`)으로 만들어 두고, 같은 프로그램으로 여러 입력을 맞춘다. 컴파일 결과와 작업 공간이 모두 호출자의 슬라이스이므로 할당이 없다.

흔한 정규식 엔진은 갈림길에서 한 길을 끝까지 가 보고 막히면 되돌아오는 **역추적** 방식이라, 어떤 패턴과 입력에서는 시간이 입력 길이에 대해 지수로 늘어난다(ReDoS). `regex` 는 모든 갈림길을 동시에 한 걸음씩 진행하는 Pike VM 방식이라 시간이 입력 길이에 비례한다. 대신 역참조처럼 역추적을 요구하는 기능은 없다. 무엇을 하지 않는지가 모듈 문서의 첫머리에 적혀 있다.

33.6 흔한 실수

> **반례. 접두사가 있었는지 묻지 않고 뗀 결과를 믿는다**

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 ✘ 접두사가 있었는지 묻지 않고 뗀 결과를 믿는다 --- 없으면 원본 그대로 돌아온다
fn path_len input line slice u8 . output u64 .
do
  let rest slice u8 be strings.remove_prefix line "GET " .
  return len rest .
end
```

실행 결과

```
$ 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` 는 접두사가 없으면 **원본 그대로** 돌려준다. 실패를 알리지 않는 대신 결과가 언제나 쓸 수 있는 뷰가 되게 한 설계다. 그래서 `POST /a` 를 받으면 "GET " 을 뗀 것처럼 보이는 일곱 바이트가 돌아온다. 접두사가 있어야 뜻이 서는 자리라면 `starts_with` 로 먼저 묻는다.

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 먼저 묻고, 없으면 따로 답한다
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
```

실행 결과

```
$ 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]
```

반례. 부호화의 결과를 출력 버퍼 통째로 여긴다

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 ✘ 결과를 `dst` 통째로 여기면 뒤에 쓰지 않은 네 바이트가 붙는다 --- 쓴 수는 12, 버
퍼는 16
  return add (mul (len dst) 100) (some_value n) .
end
```

실행 결과

```
$ lowentc --run sizes mistake_wholedst.low 0
sizes(0) = 1612
```

`codec.hex_enc` 는 16 바이트 버퍼에 12 바이트를 쓰고 12 를 돌려준다(결과 1612 는 버퍼 16 과 쓴 수 12 를 한데 적은 것이다). `dst` 통째를 결과로 쓰면 쓰지 않은 네 바이트가 뒤에 붙는다. C 에서 `sprintf` 의 반환값을 버리고 버퍼 전체를 보내는 실수와 같다. 결과는 언제나 `subslice dst 0 n` 이다.

흔한 오해. `len` 은 글자 수이고, 바이트 자리가 곧 글자 자리다

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` 은 바이트 수다 --- "안녕" 은 6 바이트, 2 글자
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 글자는 바이트 1 에서 시작하지 않는다 --- 둘째 글자는 바이트 3 에서 시작한다
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
```

실행 결과

```
$ 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` 은 바이트 수다. “안녕” 은 6 바이트, 2 글자라서 `lengths` 가 62 를 낸다. 바이트 1 은 첫 글자의 가운데라 `utf8.decode` 가 `none`(여기서 0)을 주고, 둘째 글자 “녕”(U+B155, 45397)은 바이트 3 에서 시작한다. 글자를 세거나 옮겨 다닐 때는 `utf8` 의 op 을 쓰고, 화면에서 차지하는 칸 수는 또 다르다(37장).

33.7 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`strings.starts_with s p` · `strings.find hay needle from`	묻기 · 찾기(`option u64`)	뷰 위에서 — 복사가 없다
`strings.remove_prefix s p`	접두사를 뗀 뷰 — 없으면 원본 그대로	실패 없이 늘 쓸 수 있는 뷰
`fmt.put_str buf pos s` · `fmt.put_u64 buf pos n`	호출자의 버퍼에 조립 — 다음 자리를 `option` 으로	전량 아니면 무
`utf8.count_chars s` · `utf8.decode s at`	글자 수 · 그 자리의 코드포인트	올바르지 않으면 치환 문자 대신 `none`
`codec.hex_enc src dst` · `b64_enc`	옮겨 적고 쓴 수를 준다	결과는 `subslice dst 0 n` — 암호화가 아니다
`hash.bucket_of key n` · `hash.crc data` · `hash.digest`	칸 고르기 · 손상 검출 · 변조 검출	물음마다 다른 해시
`regex`	한 번 컴파일해 여러 입력에 맞춘다	역추적하지 않는다 — 시간이 입력 길이에 비례

표 33.2 — 글자 모듈의 모양 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`strings` 는 복사 없는 뷰를, `fmt` 는 호출자의 버퍼에 조립하고 다음 자리를 `option` 으로 돌려주며, `strbuf` 는 소유 버퍼에 이어 붙인다. `utf8`·`utf16` 은 올바르지 않은 입력을 치환 문자 없이 `none` 으로 알리고, `unicode` 의 표는 기계적으로 뽑았다. `codec` 은 16진·base64 를 전량 아니면 무로 옮긴다. `hash` 는 칸 고르기·손상 검출·변조 검출에 각기 다른 해시를 주고, `regex` 는 역추적하지 않는다.

34 그릇과 정렬 — `sortlib` · `sortgen` · `hashmap` · `vecgen` · `spsc`

이 장의 차례

34.1 `u64` 를 정렬하고 찾는다 412
34.2 타입이 기준을 들고 온다 412
34.3 호출자의 슬라이스 위의 해시맵 413
34.4 자라는 제네릭 벡터 414
34.5 흐름 사이로 값을 넘기는 링 버퍼 415
34.6 그 밖의 그릇 415
34.7 흔한 실수 416
34.8 이 장의 문법 한눈에 419

먼저 알아야 할 것

22장 제네릭 · 비교는 값이 아니라 타입이 들고 온다
20장 할당기와 고정 메모리 · 할당기를 `using` 으로 건넨다
32장 표준 라이브러리의 지도 · 버퍼는 호출자의 것

돌아보기

22장에서 C 의 `qsort` 는 비교 함수를 값으로 받는데 Lowent 의 `sortgen` 은 무엇으로 받는다고 했는가?

> **답.** 비교를 **타입**이 들고 온다. 정렬할 타입이 `ordered` 트레이트를 갖추고 `less` 를 붙이면, `sort_by` 는 그 타입 전용으로 단형화되어 비교가 직접 호출로 박힌다. 이 장은 그 정렬과 함께 표준 라이브러리의 그릇들을 둘러본다.

이 장의 필요성과 맥락

정렬·탐색·해시맵·자라는 배열은 거의 모든 프로그램에 들어간다. 다른 언어에서 이런 그릇은 대개 몰래 힙에서 할당하고, 가득 차면 조용히 자란다. Lowent 의 그릇은 두 갈래다. 호출자가 뒷받침 슬라이스를 건네는 **고정 그릇**(`hashmap`·`sortlib`)과, 할당기를 `using` 으로 받아 스스로 자라는 **제네릭 그릇**(`vecgen`·`mapgen`)이다. 어느 쪽이든 메모리가 어디서 오는지가 머리에 보인다.

이 장이 끝나면

`sortlib.sort` 와 `searchlib.bsearch` 로 `u64` 슬라이스를 정렬·탐색하고, `sortgen.sort_by` 로 구조체를 타입이 정한 기준으로 정렬하는 법을 익힌다. `hashmap` 이 호출자의 슬라이스 위에서 동작하는 방식(빈칸·묘비)을 알게 된다. `vecgen` 이 할당기를 받아 자라는 모습과 그 효과가 할당기 타입에 따라 달라지는 것을 확인한다. 락 없이 흐름 사이로 값을 넘기는 `spsc` 와 그 밖의 그릇의 자리도 보게 된다.

이 장에서 답할 질문

1 `less a a` 가 참이면 어떻게 되는가?

34.1 u64 를 정렬하고 찾는다

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
```

실행 결과

```
$ lowentc --run main sorted.low
main() = 32
```

- `alloc_bytes al capacity 40` 으로 40 바이트를 받고 `view_array u64` 로 u64 다섯 칸의 슬라이스로 본다. 할당은 여기서만 일어난다.
- `sortlib.sort xs` 는 제자리에서 오름차순으로 정렬한다. `effects none` 이다.
- `searchlib.bsearch xs 19` 는 정렬된 슬라이스에서 19 의 자리를 `option` 으로 준다. 없으면 `none` 이다. `lower_bound` 는 넣을 자리를 준다.

가장 작은 값 3 과 19 의 자리 2 로 32 가 나온다. `bsearch` 는 입력이 정렬되어 있다고 **믿는다**. 정렬되지 않은 슬라이스를 주면 틀린 답을 낸다. 그 믿음은 모듈 문서에 적힌 전제다.

34.2 타입이 기준을 들고 온다

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
```

실행 결과

```
$ lowentc --run main rows.low
main() = 231
```

`score` 는 `satisfies sortgen.ordered .` 로 순서를 안다고 선언하고, `score.less` 가 "점수가 높은 것이 앞" 이라는 기준을 준다. `sortgen.sort_by score rs` 는 그 기준으로 정렬한다. 점수 95·80·70 의 번호가 차례로 2·3·1 이다.

내림차순이나 여러 키로 정렬하고 싶으면 `less` 를 그렇게 쓰면 된다. 모드 인자를 다는 대신 타입이 뜻을 들고 온다. `sort_by` 는 삽입정렬이라 **안정**하고(같은 점수의 차례가 바뀌지 않는다) 거의 정렬된 입력에 빠르다. 큰 배열에는 `sort_fast`(제네릭 quicksort) 가 있다. `u64` 같은 스칼라는 트레이트를 갖출 수 없으므로 칸 하나짜리 구조체로 감싼다. 배치는 그대로 8 바이트다.

> **문.** `less a a` **가 참이면 어떻게 되는가?**
>
> **답.** 정렬이 멈추지 않거나 틀린 차례를 낼 수 있다. `less` 는 **엄격한 약한 순서**여야 한다 — 특히 자기 자신보다 앞일 수 없다. `le` 로 적어야 할 자리를 `lt` 로, 또는 그 반대로 적는 실수가 여기서 난다. 트레이트는 op 이 **있다**는 것을 검사하지 그 op 이 수학적 성질을 지키는지까지 검사하지는 않는다. 그 성질은 모듈 문서에 적힌 계약이다.

34.3 호출자의 슬라이스 위의 해시맵

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
```

실행 결과

```
$ lowentc --run main table.low
main() = 78
```

hashmap 은 u64 → u64 오픈 어드레싱 해시맵이다. 뒷받침은 호출자가 드는 mut slice u64 이고, [키0+1, 값0, 키1+1, 값1, …] 으로 배치된다. size 는 칸 수(len / 2)다. 처음에는 전부 0(빈칸)이어야 하는데, alloc_bytes 가 0 으로 채운 바이트를 주므로 그대로 쓴다.

- put slots 7 700 은 넣고, 같은 키로 다시 put 하면 값을 바꾼다. 가득 차면 false 다.
- lookup 은 option 을 준다.
- del 은 칸을 빈칸으로 되돌리지 않고 **묘비**로 덮는다. 빈칸으로 되돌리면 그 뒤에 이어진 탐색 사슬이 끊겨 뒤의 키를 못 찾기 때문이다.

이 설계의 대가도 문서에 적혀 있다. 빈칸(0)과 묘비를 표시하려고 가장 큰 키 둘을 쓸 수 없고, 자동으로 자라지 않는다. 바이트열 키는 strmap 이, 자라는 제네릭 해시맵은 mapgen 이 맡는다.

34.4 자라는 제네릭 벡터

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
```

실행 결과

```
$ lowentc --run main growing.low
main() = 100
```

- `allocs.heap_bytes` 를 띄워 할당기로 쓴다. `cap heap` 을 쥔 `main` 만 띄울 수 있다(20장).
- `vecgen.open u32 4` 는 원소 타입 `u32`, 처음 용량 4 인 벡터를 연다. 할당기는 바인딩의 `using hb` 로 건넨다.
- `vecgen.append … v x` 는 자리가 모자라면 할당기에게 더 청해 자란다. 실패하면 `false` 다. 벡터는 자기 할당기를 이미 들고 있으므로 `append` 에는 `using` 을 적지 않는다. 적으면 `E-ALLOC-USING-UNUSED` 로 거절된다.
- `vecgen.at … v 50` 은 50 번 원소를 `option` 으로 준다. $50 \times 2 = 100$ 이다.

`vecgen.append` 의 효과는 `state via a` 다. 여기서 `a` 가 `heap_bytes` 이므로 이 인스턴스의 효과는 `heap state` 이고, `main` 의 머리에도 `heap` 이 선다. 같은 코드를 빌린 바이트 위의 범프로 열면 `state` 만 선다. 그릇이 운영체제 없는 기계에서 도는지를 할당기 타입이 정한다.

> **흔한 오해. 자라는 벡터는 결국 숨은 할당이다**
>
> 할당은 일어나지만 숨지 않는다. `append` 의 효과 줄이 할당기의 효과를 그대로 싣고, 부르는 op 의 머리에 `heap` 이나 `alloc` 이 선다. 할당기가 모자라면 `append` 가 `false` 를 준다 — 메모리 부족이 값이다. 숨은 할당은 머리에 보이지 않고 실패하면 프로그램을 멈추는 할당이다.

34.5 흐름 사이로 값을 넘기는 링 버퍼

`spsc` 는 락 없는 단일 생산자·단일 소비자 링 버퍼다. 한 흐름이 넣고 다른 한 흐름이 뺀다. 제어 칸과 뒷받침 칸을 호출자가 건네고, 넣고 빼는 op 은 `cap atomic` 을 받아 원자 연산으로 커서를 옮긴다(27장). 인터럽트 처리기와 보통 코드가 값을 주고받는 자리(30장)에 쓰인다.

락 없는 자료구조는 드문 차례에서만 틀리기로 악명이 높다. `spsc` 의 정확성은 약한 메모리 모델에서 이미 증명된 알고리즘의 증명을 **빌려** 확인했다. 다중 생산자·다중 소비자 큐나 seqlock 은 빌릴 증명이 없어 넣지 않았다(47장).

34.6 그 밖의 그릇

모듈	한 줄
`strmap`	바이트열 → `u64` 해시맵. 키 바이트도 호출자의 슬라이스에 담는다
`mapgen`	제네릭 해시맵 `table k v`. 스스로 다시 뿌린다
`vecs · growvec`	자라는 바이트 벡터. `growvec` 은 `vecgen.vec u8` 의 짧은 이름이다
`nodelist`	고정 크기 침입형(intrusive) 목록
`soa`	구조체 배열을 칸별 배열로 두는 배치의 실험

표 34.1 — 그 밖의 그릇 모듈

34.7 흔한 실수

반례. 정렬하지 않은 슬라이스를 이진 탐색한다

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 ✘ 정렬하지 않고 이진 탐색한다 --- 42 가 있는데 못 찾는다
  let at option u64 be searchlib.bsearch xs 42 .
  guard is_some at . else return 99 .
  return some_value at .
end
```

실행 결과

```
$ lowentc --run find42 mistake_unsorted.low 0
find42(0) = 99
```

42 는 0 번 칸에 있지만 `bsearch` 는 99(못 찾음)를 낸다. 이진 탐색은 가운데 값과 견주어 절반을 버리는데, 정렬되지 않았으면 버린 절반에 답이 있을 수 있다. 멈추지도 경고하지도 않으므로 가장 찾기 어려운 종류의 결함이다. 운이 좋으면 찾기도 한다(같은 슬라이스에서 19 는 찾는다). 이진 탐색 앞에는 언제나 `sortlib.sort` 가 있어야 하고, 정렬된 상태를 지키는 코드가 흩어져 있다면 `lower_bound` 로 넣을 자리를 찾아 정렬을 유지한다.

반례. `less` 를 `ge` 처럼 엄격하지 않게 적는다

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 ✘ "앞" 을 `ge` 로 적었다 --- 같은 점수끼리도 서로 앞이 된다
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
```

실행 결과

```
$ lowentc --run order_strict mistake_lessge.low 0
order_strict(0) = 123
$ lowentc --run order_loose mistake_lessge.low 0
order_loose(0) = 321
```

세 점수가 모두 70 이다. `gt` 로 적은 `strict_score` 는 넣은 차례 1·2·3 을 지켜 123 을 내고, `ge` 로 적은 `loose_score` 는 같은 점수끼리도 "앞" 이라 서로 자리를 바꿔 321 을 낸다. 안정 정렬이 약속한 "같은 것의 차례가 바뀌지 않는다" 가 깨진 것이다. 다른 정렬 알고리즘에서는 멈추지 않거나 틀린 차례를 낼 수도 있다. 트레이트는 `less` 가 **있는지**만 검사하므로, "자기 자신보다 앞일 수 없다" 는 성질은 적는 사람이 지킨다.

반례. 이미 할당기를 든 그릇에 `using` 을 또 적는다

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 ✘ 벡터가 이미 할당기를 들고 있는데 `using` 을 또 적었다
    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
```

실행 결과

```
$ 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` 은 할당기를 받아 벡터 안에 넣어 둔다. 그 뒤의 `append` 는 벡터가 든 할당기를 쓰므로 바인딩의 `using hb` 는 아무 뜻이 없다. 뜻 없는 표시는 "이 호출이 `hb` 에서 깎는다" 는 거짓 정보가 되므로 `E-ALLOC-USING-UNUSED` 로 거절된다. `using` 은 할당기를 **처음 받는** 자리(`open`)에만 적는다.

흔한 오해. `hashmap` 에는 어떤 `u64` 든 키로 넣을 수 있다

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 가장 큰 키 둘은 빈칸·묘비 표시에 쓰이므로 넣을 수 없다
  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
```

실행 결과

```
$ lowentc --run main hashmap_topkeys.low
main() = 4
```

결과 4 는 셋째 `put` 만 성공했다는 뜻이다. 가장 큰 키 둘(`18446744073709551615`·`18446744073709551614`)은 빈 칸과 묘비를 표시하는 데 쓰이므로 넣을 수 없고 `put` 이 `false` 를 준다. 이 대가는 모듈 문서 첫머리에 적혀 있다. 키가 전 범위를 쓸 수 있으면 키를 구조체에 담아 `mapgen` 을 쓴다.

34.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`sortlib.sort xs` · `searchlib.bsearch xs k`	`u64` 제자리 정렬 · 정렬된 입력에서 탐색(`option`)	할당하지 않는다 — 탐색은 정렬을 믿는다
`struct score do satisfies sortgen.ordered . … end` + `fn score.less`	타입이 정렬 기준을 들고 온다	모드 인자 대신 타입 — `less` 는 엄격하게
`sortgen.sort_by score rs` · `sort_fast`	안정 삽입정렬 · 큰 배열용 quicksort	고르는 기준이 이름에 있다
`hashmap.put slots k v` · `lookup` · `del`	호출자 슬라이스 위의 `u64 → u64` 맵	가득 차면 `false` — 삭제는 묘비
`let vo … using hb be vecgen.open u32 4 .`	할당기를 받아 자라는 벡터를 연다	`using` 은 처음 받는 자리에만
`vecgen.append u32 allocs.heap_bytes v x`	자라며 넣는다 — 실패하면 `false`	효과가 할당기 타입을 따라간다 (`state via a`)
`spsc`	락 없는 단일 생산자·단일 소비자 링 버퍼	원자 연산 — 증명을 빌렸다

표 34.2 — 그릇과 정렬의 모양 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`sortlib·searchlib` 는 `u64` 슬라이스를 제자리에서 정렬·탐색하고, `sortgen` 은 `ordered` 를 갖춘 타입이 기준을 들고 온다. `hashmap·strmap` 은 호출자의 슬라이스 위에서 묘비로 삭제하는 고정 그릇이다. `vecgen·mapgen` 은 할당기를 `using` 으로 받아 자라며, 효과가 할당기 타입을 따라간다. `spsc` 는 원자 연산으로 흐름 사이에 값을 넘기고 그 정확성은 빌린 증명으로 확인했다.

35 저장소와 핸들 — `pool`·`shard`·`budget`·`wire`

이 장의 차례

35.1 세대 핸들 421
35.2 브랜드 — 풀을 섞지 못하게 422
35.3 겹치지 않는 조각 — `shard` 422
35.4 비트 예산 — `budget` 423
35.5 비트 칸 — `wire` 424
35.6 그 밖의 저장소 모듈 424
35.7 흔한 실수 425
35.8 이 장의 문법 한눈에 427

먼저 알아야 할 것

10장 묶음 · 번호로 이으면 옛 번호가 엉뚱한 노드를 본다
13장 이름 붙인 타입 · `newtype` 은 표현이 같은 새 타입이다
19장 소유 · `owned` 값은 넘기면 옮겨진다
20장 할당기와 고정 메모리 · 범프는 마지막 조각 하나만 돌려받는다

돌아보기

10장에서 나무를 번호로 이을 때 생기는 불편으로 무엇을 들었고, 그것을 무엇이 잡는다고 했는가?

> **답.** 노드를 지우고 그 번호를 다시 쓰면 옛 번호를 들고 있던 쪽이 엉뚱한 노드를 본다는 것이었다. 표준 라이브러리의 `pool` 이 번호에 세대 수를 함께 담는 **세대 핸들**로 이것을 실행 중에 잡는다고 했다. 이 장이 그 `pool` 과, 저장소를 다루는 다른 모듈들을 본다.

이 장의 필요성과 맥락

범프 할당기와 영역은 "한꺼번에 돌려준다" 에 맞는 도구다. 게임 객체, 연결 목록, 캐시 페이지처럼 만들고 지우기를 끝없이 되풀이하는 데이터에는 맞지 않는다. 그런 데이터에 포인터를 쓰면 지운 뒤 사용이 생기고, 번호를 쓰면 번호 재사용이 생긴다. 이 장의 모듈들은 같은 문제를 다른 층에서 막는다 — 실행 중 세대 비교, 타입의 브랜드, 소유 토큰, 번역 시점의 비트 예산. 새 낱말은 하나도 늘지 않았다. 모두 앞의 장들에서 배운 규칙의 조합이다.

이 장이 끝나면

`pool` 이 고정 크기 블록을 세대 핸들로 빌려주고, 돌려준 뒤의 옛 핸들을 알아보는 방식을 익힌다. `newtype` 브랜드로 서로 다른 풀을 섞는 실수를 번역에서 막는 법을 알게 된다. `shard` 가 소유 토큰으로 저장소를 겹치지 않는 조각으로 나누는 모습, `budget` 이 핸들의 비트 예산을 번역 시점에 확인하는 모습, `wire` 가 마스크 하나로 비트 칸을 다루는 모습도 보게 된다.

이 장에서 답할 질문

1 세대 비교는 실행 중 검사다. 번역에서 막을 수는 없는가?

35.1 세대 핸들

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
```

실행 결과

```
$ lowentc --run main blocks.low
main() = 3
```

- `pool.init objects mem gens 16` 은 호출자가 건넨 바이트 `mem` 을 16 바이트 블록으로 나누고, 블록마다 세대 수를 `gens` 에 둔다. 풀은 `mem` 과 `gens` 를 봉해 들어서 이후의 op 은 그것을 따로 받지 않는다.
- `pool.take` 는 블록 하나를 빌려주고 **핸들**을 준다. 핸들에는 블록 번호와 그때의 세대 수가 들어 있다.
- `pool.release` 는 블록을 돌려받으며 그 블록의 세대 수를 하나 올린다.
- 돌려준 뒤의 옛 핸들은 세대가 달라 `alive` 가 거짓이고, 두 번째 `release` 도 거절된다.

결과 3 은 “돌려주기 전에 살아 있었고(1), 첫 반납이 성공했다(2)” 이고, 돌려준 뒤 살아 있지도(4) 두 번째 반납이 성공하지도(8) 않았다는 뜻이다. 지운 뒤 사용과 두 번 해제가 실행 중에 **값으로** 드러난다.

> **문. 세대 비교는 실행 중 검사다. 번역에서 막을 수는 없는가?**
>
> **답.** 핸들은 복사되고 저장되고 파일에 쓰일 수 있는 값이라서, 어느 핸들이 이미 낡았는지를 번역이 모두 알 수는 없다. 그래서 이 자리는 동적 검사다(19장의 강도 표에서 “세대 핸들의 늘어진 참조 — 동

> 적”). 대신 번역이 막을 수 있는 실수 — 풀을 섞는 것 — 은 아래처럼 번역에서 막는다. 무엇을 번역이 막고 무엇을 실행이 막는지가 모듈 문서의 첫머리에 적혀 있다.

35.2 브랜드 — 풀을 섞지 못하게

풀과 핸들은 번역 시점 타입 매개변수로 **브랜드**를 든다. `newtype objects u8 .` 선언 하나가 곧 저장소 하나다.

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
```

실행 결과

```
$ 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)
```

`sounds` 풀에서 빌린 핸들을 `meshes` 풀에 돌려주려 하면, 두 인스턴스가 서로 다른 구체 타입이라 거절된다. 브랜드는 값이 아니라 타입이라서 핸들의 칸은 늘지 않는다. 비용이 0 이다. 같은 브랜드로 저장소를 두 번 여는 것도 거절된다(`E-BRAND-REUSED`) — 그러면 한 이름이 둘을 가리키게 되어 브랜드가 막으려던 혼동이 돌아온다.

35.3 겹치지 않는 조각 — shard

`shard` 는 저장소 한 덩어리를 서로 겹치지 않는 조각으로 나누고, 조각마다 **토큰**을 준다. 토큰은 “이 범위는 내 것” 이라는 권리이고 `owned` 값이다.

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
```

실행 결과

```
$ 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` 는 전체 토큰 `r` 을 **소비해** 둘로 나눈다. 그 뒤 `r` 로 폭을 물으면 `E-OWN-MOVED` 다. 이 실수를 막는 것은 `shard` 모듈이 아니라 **언어**다. 토큰이 `owned` 라서 넘기는 순간 손을 떠난다. 접근 단위라는 개념이 새 규칙이 아니라 이미 있던 규칙의 결과다. 나눈 조각을 여러 흐름에 태우든 순차로 돌든, 겹치지 않는다는 사실 자체가 값이다.

같은 모양이 `pagecache` 에도 있다. 페이지를 고정(`acquire`)하면 캐시 토큰이 소비되어, 고정이 살아 있는 동안 축출(`evict`)에 건넬 토큰이 손에 없다. 고정된 페이지를 쫓아내는 결함이 번역에서 막힌다.

35.4 비트 예산 — `budget`

세대 핸들을 한 워드에 넣으려면 "칸 번호·샤드 번호·세대 수에 몇 비트씩 주나" 를 정해야 한다. `budget` 은 그 수를 소스에 적고 도구가 지키게 한다.

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
```

실행 결과

```
$ 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 …` 은 합이 72 비트라서 64 비트 워드에 들어가지 않는다. 인자가 모두 상수이므로 계약 위반이 번역에서 판정된다 (14장). 진단이 적은 사연대로, 맞지 않는 비트 예산을 지닌 프로그램이 한때 실행할 수 있는 모습으로 배포되었다. 이 모듈은 그 일 때문에 생긴 검사를 쓰는 첫 자리다. 세대 수가 다 차면 그 칸은 **은퇴**한다(`retired`). 몰래 0 으로 감아 옛 핸들이 다시 살아나는 일은 없다.

35.5 비트 칸 — `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
```

실행 결과

```
$ lowentc --run mode_of fields.low 1234
mode_of(1234) = 4
$ lowentc --run with_mode fields.low 1234 5
with_mode(1234, 5) = 1490
```

한 정수 안에 여러 값을 나란히 넣을 때, `wire` 는 칸을 **마스크 하나**로 다룬다. 마스크 `0xF00`(3840)는 "8번 비트에서 시작하는 4 비트" 라는 자리와 폭을 둘 다 말한다. `pick` 은 그 칸의 값을 꺼내고, `merge` 는 그 칸만 갈아 끼운다. 1234 는 `0x4D2` 이므로 칸의 값은 4 이고, 5 로 갈아 끼우면 `0x5D2`(1490)다. 손으로 적을 때 늘 갈리던 "자리" 와 "폭" 두 수가 한 마스크에서 나오므로 갈릴 수 없다. 이름 붙은 켬/끔 설정을 한 워드에 담는 일은 `flags` 가 맡는다.

> **흔한 오해. 이런 모듈은 결국 unsafe 코드로 짜여 있을 것이다**
>
> `pool`·`shard`·`budget`·`wire` 는 모두 이 책에서 배운 Lowent 로 쓰였고 `unsafe` 가 없다. 세대 비교는 슬라이스 색인과 비교이고, 브랜드는 `newtype` 과 타입 매개변수이고, 토큰은 `owned` 이고, 예산은 계약이다. 새 낱말도 새 내장 연산도 늘지 않았다. 라이브러리가 특권 없이 쓰였기 때문에, 그 라이브러리를 쓰는 프로그램의 보장이 끊기지 않는다.

35.6 그 밖의 저장소 모듈

모듈	한 줄

allocs	범프 할당기와 할당기 트레이트, 기본 할당기(20장)
segarena	같은 크기의 조각으로 자라며 이미 있는 것을 옮기지 않는다. 비용이 op 이름에 적혀 있다
pagecache	오래 사는 이름(페이지 번호)과 짧게 닿는 권리(고정)를 가른다
segview	흩어진 조각 뷰의 커서·총길이·펴기
lifemode · lifeatom	값의 끝을 정하는 네 방법 — 단일 소유·스레드 국한 참조 계수·원자 참조 계수·외부 완결

표 35.1 — 그 밖의 저장소 모듈

35.7 흔한 실수

반례. 돌려준 블록의 바이트를 묻지 않고 꺼낸다

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 ✘ 돌려준 블록의 바이트를 묻지 않고 꺼낸다 --- 세대가 달라 `none` 이다
  var b mut slice u8 be some_value (pool.bytes objects p h) .
  set (index b 0) 7 .
  return index b 0 .
end
```

실행 결과

```
$ lowentc --run main mistake_staleuse.low
== ir diagnostics (1) ==
0:0 E-VM-NONE: some_value of none (panic)
```

pool.bytes 는 블록 바이트에 닿는 **유일한 문**이고, 세대가 맞지 않으면 none 을 준다. C 라면 해제한 블록에 7 을 쓰는 순간 다른 객체가 조용히 망가지지만, 여기서는 그 자리가 none 이라는 값으로 드러난다.

그 값을 `some_value` 로 묻지 않고 꺼내면 이 예제처럼 멈춘다. 멈추는 편이 망가지는 것보다 낫지만, 핸들이 오래 살 수 있는 코드는 `is_some` 으로 묻고 낡은 핸들을 따로 다룬다.

반례. 같은 브랜드로 저장소를 둘 연다

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 ✘ 같은 브랜드 `objects` 로 저장소를 또 연다 --- 두 풀의 핸들이 섞여도 타입이 막지
못한다
  let p2 option (pool.block_pool objects) be pool.init objects (some_value g2) gens2 16
.
  return 0 .
end
```

실행 결과

```
$ 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)
```

브랜드 `objects` 는 저장소 **하나**의 이름이다. 둘째 `pool.init objects …` 가 통과하면 두 풀이 같은 타입을 나누게 되어, 첫 풀의 핸들을 둘째 풀에 돌려주는 실수를 타입이 막지 못한다. 그래서 `E-BRAND-REUSED` 다. 저장소마다 `newtype` 을 하나씩 선언한다. 선언 한 줄이 비용 없는 구별이다.

흔한 오해. `wire` 의 칸에 넣은 값은 넘치면 멈춘다

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 칸은 네 비트(0 … 15)다 --- 20 은 잘려 4 가 들어간다
fn put_mode input m u64 . output u64 .
do
  return wire.put 3840 m .
end

rem 잘리는 것이 싫으면 먼저 묻는다
fn fits_mode input m u64 . output bool .
do
  return wire.fits 3840 m .
end
```

실행 결과

```
$ 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` 은 칸보다 큰 값을 **자른다**. 네 비트 칸에 20(`0b10100`)을 넣으면 아래 네 비트 4 만 들어가 1024(`0x400`)가 된다. 모듈 문서가 그렇게 약속하고, 잘리는 것이 싫으면 `wire.fits` 로 먼저 물으라고 적는다(20 은 0, 곧 거짓). 선으로 나가는 비트 칸은 잘리는 것이 표준 동작인 경우가 많아 이렇게 정했다. 멈추기를 원하는 자리는 `requires wire.fits …` 를 계약으로 적는다.

35.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`newtype objects u8 .`	저장소 하나의 브랜드	비용 없는 구별 — 섞으면 `E-TYPE-INSTANCE`, 두 번 열면 `E-BRAND-REUSED`
`pool.init objects mem gens 16`	호출자 바이트를 블록으로 나눈 풀(봉해 든다)	뒤의 op 은 `mem·gens` 를 따로 받지 않는다
`pool.take · pool.release · pool.alive`	빌리기(세대 핸들) · 돌려주기(세대 올림) · 묻기	지운 뒤 사용·두 번 해제가 값으로 드러난다
`pool.bytes objects p h → option mut slice u8`	블록 바이트에 닿는 유일한 문	세대가 다르면 `none`
`var r owned shard.token grid be shard.open grid 8 .`	겹치지 않는 조각의 소유 토큰	나누면 옛 토큰은 `E-OWN-MOVED`
`budget.pack …`	비트 예산을 계약으로 확인	맞지 않으면 번역에서 `E-CONTRACT-IMPOSSIBLE`
`wire.pick mask w · wire.merge mask w v · wire.fits`	마스크 하나로 비트 칸 읽기 · 갈아 끼우기 · 들어가는지	자리와 폭이 한 수에서 나온다 — `put` 은 자른다

표 35.2 — 저장소와 핸들의 모양 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`pool` 은 세대 핸들로 블록을 빌려주고, 돌려준 뒤의 옛 핸들과 두 번 해제를 실행 중에 값으로 알린다. `newtype` 브랜드는 풀을 섞는 실수를 번역에서 막고 비용이 없다. `shard·pagecache` 는 `owned` 토큰으로 겹침과 고정 중 축출을 번역에서 막는다. `budget` 은 핸들의 비트 예산을 계약으로 번역 시점에 확인하고, `wire` 는 마스크 하나로 비트 칸의 자리와 폭을 함께 다룬다. 모두 새 낱말 없이 언어의 규칙만으로 쓰였다.

36 입출력·네트워크·시간·난수·암호

이 장의 차례

36.1 버퍼링 출력 — 비우기를 잊으면 번역이 거절한다 430
36.2 네트워크 — 핸들은 자원이다 431
36.3 난수 — 재현되는 열과 운영체제 엔트로피 432
36.4 시간 — 단조 시계와 벽시계 432
36.5 HTTP 요청 파서 — 알맹이는 거절이다 433
36.6 암호 — 쌓인 차례와 아직 없는 꼭대기 434
36.7 흔한 실수 434
36.8 이 장의 문법 한눈에 437

먼저 알아야 할 것

28장 입출력과 파일 · 파일 핸들은 완결이 필요한 소유 값이다
16장 권한 · `cap net·cap clock·cap random`
17장 실패를 설계하기 · HTTP 파서의 알맹이는 거절이다

돌아보기

28장에서 `files.read` 의 답이 세 자리로 갈린 까닭은 무엇이었는가?

> **답.** 값 하나가 “끝” 과 “실패” 두 뜻을 나르면, 읽기 실패가 “다 읽었다” 로 보고되기 때문이었다. 실제로 줄 세기 프로그램이 읽기 실패를 “0 줄” 이라는 성공으로 냈다. 이 장의 모듈들도 같은 원칙 — 뜻마다 한 자리, 권한마다 한 종류 — 을 따른다.

이 장의 필요성과 맥락

바깥과 닿는 모듈은 두 가지를 섞기 쉽다. **셈**과 **권위**다. 출력할 글자를 짓는 것과 표준출력에 내보내는 것, 난수의 다음 값을 계산하는 것과 운영체제에서 엔트로피를 얻는 것, 규약의 바이트를 해석하는 것과 연결을 여는 것은 서로 다른 일이다. 섞으면 순수한 계산까지 권한과 효과를 끌고 다니고, 시험하기 어려워진다. 이 장의 모듈들은 둘을 가른다. 그리고 암호 모듈들이 TLS 를 향해 어떤 차례로 쌓여 있는지를 본다.

이 장이 끝나면

`outbuf` 로 출력을 버퍼에 모았다가 내보내고, 비우지 않으면 번역이 거절한다는 것을 확인한다. `net` 의 프로세스 안 연결 쌍으로 바이트를 주고받는 법, 결정적인 `random.step` 과 운영체제 엔트로피 `random.bytes` 를 가르는 이유, 단조 시계와 벽시계의 약속이 다르다는 것을 알게 된다. HTTP 요청 파서가 무엇을 거절하는지, 암호 모듈이 어떤 층으로 쌓여 있고 무엇이 아직 없는지도 보게 된다.

이 장에서 답할 질문

1 예제가 실제 TCP 포트를 열지 않고 프로세스 안 쌍을 쓰는 이유는 무엇인가?

36.1 버퍼링 출력 — 비우기를 잊으면 번역이 거절한다

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
```

실행 결과

```
$ lowentc --run main buffered.low
buffered output
main() = 0
```

- `outbuf.buf_open 1` 은 표준출력(1)에 내보낼 **대기 중 출력**을 만든다. `owned outbuf.pending` 이다.
- `outbuf.buf_write out p buf s` 는 `s` 를 버퍼에 모으고, 버퍼가 차면 내보낸 뒤 이어 쓴다. 대기 값을 받아 새 대기 값을 돌려준다(소유가 옮겨 다닌다).
- `outbuf.buf_finish` 가 남은 바이트를 내보내고 대기 값을 끝낸다.

버퍼가 8 바이트뿐이라 “buffered “ 를 쓰는 도중에 한 번 비워진다. 출력은 같다. 한 바이트씩 `write_out` 하는 대신 모아서 내보내는 것이 버퍼링의 값이다. 마지막 `buf_finish` 를 잊으면 버퍼에 남은 바이트가 사라진다. 그래서 번역이 거절한다.

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
```

실행 결과

```
$ 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` 가 `owned pending` 을 받아 `result` 를 돌려주므로 `pending` 은 완결이 필요한 타입이다(19장). "마지막 줄이 안 찍혔다" 는 결함이 실행 전에 드러난다.

36.2 네트워크 — 핸들은 자원이다

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
```

실행 결과

```
$ lowentc --run main loopback.low
main() = 4
```

`net.pair_of` 는 프로세스 안에서 서로 이어진 연결 한 쌍을 만든다. 한쪽(`pair_a`)으로 "ping" 을 보내고 다른 쪽(`pair_b`)에서 받아 4 바이트를 얻는다. `net.shut_pair` 로 쌍을 닫는다. 닫기는 실패할 수 있고 `result` 를 준다.

`net` 의 연결과 리스너는 `files` 의 핸들처럼 **자원**이다. 소켓을 닫지 않으면 서버가 오래 돌수록 기술자가 바닥난다. `serve·dial·take` 로 TCP 루프백 연결을 열 수 있고, 모든 op 이 `cap net` 을 첫 인자로 받는다. 이 모듈을 쓰지 않는 코드는 네트워크에 닿을 수 없다는 것이 머리에 보인다.

> **문. 예제가 실제 TCP 포트를 열지 않고 프로세스 안 쌍을 쓰는 이유는 무엇인가?**
>
> **답.** 예제는 VM 과 네이티브에서 똑같이 돌아 같은 답을 내야 한다. 실제 포트는 기계의 상태(이미 쓰이는 포트, 방화벽)에 따라 결과가 달라진다. 프로세스 안 쌍은 바깥 상태에 기대지 않으므로 결정적이다. 실제 서버 코드는 `serve` 와 `take` 로 같은 `send_all·recv_once` 를 쓴다.

36.3 난수 — 재현되는 열과 운영체제 엔트로피

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
```

실행 결과

```
$ 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` 은 시드에서 다음 값을 **계산**한다. 권한도 효과도 없는 순수한 `fn` 이다. 같은 시드 42 로 두 번 굴리면 두 번 다 3 이다. 시뮬레이션·시험·절차적 생성처럼 **재현되어야 하는** 난수는 이것을 쓴다.

운영체제의 엔트로피는 `random.bytes k dst` 로 얻고 `cap random` 을 받는다. 키와 논스처럼 **예측되면 안 되는** 난수는 이쪽이다. 두 일에 서로 다른 이름과 권한을 주는 까닭은, 하나로 합치면 재현되어야 할 시험이 운영체제 엔트로피에 기대게 되거나 반대로 키가 예측 가능한 열에서 나오기 때문이다. `below_biased` 는 이름대로 치우침이 있는 범위 줄이기다. 치우침이 문제인 자리에서는 쓰지 않는다는 뜻이 이름에 들어 있다.

36.4 시간 — 단조 시계와 벽시계

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
```

실행 결과

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

`clock.now_ns` 는 **단조** 시계다. 뒤로 가지 않으므로 두 번 재서 뺄 수 있다(`since_ns`). 절대 시각이 아니다. 벽시계(`local_packed` 와 `year_of` 따위)는 사람이 쓰는 날짜와 시각이고, 시간대 변경이나 시각 동기화로 뒤로 갈 수 있다. 경과 시간을 벽시계로 재면 음수가 나올 수 있다. 둘은 약속이 다르므로 op 이 다르다.

시계를 읽으면 같은 입력에 다른 답이 나오므로 결정성이 깨진다. 그래서 `cap clock` 이 필요하다. 그러나 바깥에 흔적을 남기지는 않으므로 효과는 `none` 이다(16장). 이 예제는 실행할 때마다 답이 달라서, 이 책의 검증 스크립트는 검사만 한다. `sleep_ms` 는 기다리므로 `wait` 효과다.

36.5 HTTP 요청 파서 — 알맹이는 거절이다

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
```

실행 결과

```
$ 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` 는 요청의 메서드를 번호로(`GET` 은 1), `http.version_ok` 는 요청 줄의 판이 올바른지 답한다. 모르는 메서드 `BREW` 는 0 이다. 파서는 순수 계산이라 권한이 없다. 연결에서 바이트를 받는 일(`net`)과 그 바이트를 해석하는 일(`http`)이 다른 모듈이다.

이 파서의 설계 원칙은 **애매한 입력을 받아 주지 않는 것**이다. 요청 밀반입(request smuggling)은 서버와 프록시가 애매한 요청을 서로 다르게 받아들이는 자리에서 산다. 헤더 이름의 공백, 중복된 길이 헤더, 잘못된 줄바꿈 같은 것을 관대하게 받아 주면 그 틈이 생긴다. 이 모듈은 응답을 짓지 않는다 — 요청을 읽는 쪽이다.

> **흔한 오해. 관대한 파서가 사용자에게 친절하다**
>
> 사람이 쓰는 설정 파일이라면 그럴 수 있다. 네트워크 규약에서는 한쪽의 관대함이 다른 쪽과의 해석 차이가 되고, 그 차이가 공격면이 된다. 규약의 경계에서 친절한 것은 거절의 이유를 이름으로 말하는 것이다.

36.6 암호 — 쌓인 차례와 아직 없는 꼭대기

암호 모듈들은 모두 순수 계산(L0)이고, TLS 1.3 을 향해 층으로 쌓여 있다.

층	모듈
유도	`hash`(SHA-256) · `hmac`(HMAC-SHA256·HKDF)
봉인(스위트 둘)	`chacha·poly` → `aead` · `aes` → `gcm`
키 합의	`x25519`
서명 검증	`bigint` → `rsa`(PSS) · `p256`(ECDSA) · `ed25519`
서명 생성	`p256` → `ecdsa`(논스를 난수 없이 유도한다)
키·인증서 꺼내기	`pem` → `der`(공개키만 꺼내는 최소 파서)
규격 계산	`tls13`(키 스케줄·레코드·전사·Finished)
핸드셰이크	`tlssrv`(서버 핸드셰이크 양방향 + 응용 데이터 레코드 — 전송은 아직)

표 36.1 — 암호 모듈이 쌓인 차례

각 모듈의 문서 첫머리가 **혼자 쓰면 틀리는 방법**을 경고한다. `chacha` 는 혼자서는 안전하지 않고(→ `aead`), `aes` 는 혼자 쓰면 대개 틀리며 (→ `gcm`), `poly` 의 키는 메시지마다 새것이어야 하고, `gcm` 의 논스를 되풀이하면 안 된다. `x25519` 의 작은 위수 점 거르기는 부르는 쪽의 일이다. 암호를 쓴다면 봉인은 `aead` 에서 시작한다.

그리고 꼭대기는 **아직 없다**. `tlssrv` 는 핸드셰이크를 모두 하지만 실제 소켓 위의 전송은 아직이고, `der` 은 인증서 체계(PKI)가 아니며 인증서는 밖에서 받는다. 없는 것을 있는 척하지 않는다는 원칙이 암호에서 가장 중요하다.

36.7 흔한 실수

> **반례. `outbuf.buf_write` 에 넘긴 옛 대기 값으로 또 쓴다**

`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 ✘ `write` 가 가져간 옛 대기 값 `p` 로 또 쓴다 --- 새 대기 값은 `r1` 안에 있다
  let r2 result (owned outbuf.pending) outbuf.io_error be outbuf.buf_write out p buf "se
cond\n" .
  drop r2 .
  return 0 .
end
```

실행 결과

```
$ 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` 는 `owned pending` 을 받아 **새** 대기 값을 `result` 안에 돌려준다. 넘긴 `p` 는 이미 옮겨 갔으므로 다시 쓰면 `E-OWN-MOVED` 다. 대기 값이 옮겨 다니는 까닭은, 버퍼에 무엇이 남았는지를 아는 값이 언제나 하나뿐이어야 마지막 `buf_finish` 를 잊었는지 번역이 셀 수 있기 때문이다. 이 장의 `buffered.low` 처럼 `p` → `p2` → `p3` 으로 이어 받는다.

반례. 같은 시드에서 두 번 굴린다

examples/ch36/mistake_sameseed.low

```
module mistake_sameseed .
rem run: two_rolls 2

use random .

rem ✘ 두 번 모두 같은 시드에서 굴린다 --- 두 주사위가 늘 같다
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
```

실행 결과

```
$ lowentc --run two_rolls mistake_sameseed.low 2
two_rolls(2) = 55
```

`random.step` 은 순수한 `fn` 이라 같은 입력에 늘 같은 답을 낸다. 두 굴림에 같은 `seed` 를 주면 두 주사위가 늘 같다(시드 2 에서 55). 전역 난수 상태가 없으므로 **다음 상태**를 이어 넘기는 일은 부르는 쪽의 몫이다.

examples/ch36/sameseed_fixed.low

```
module sameseed_fixed .
rem run: two_rolls 2

use random .

rem 다음 굴림은 앞 굴림이 낸 상태에서 시작한다
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
```

실행 결과

```
$ lowentc --run two_rolls sameseed_fixed.low 2
two_rolls(2) = 56
```

고친 판은 첫 걸음이 낸 `s1` 에서 둘째 걸음을 시작해 56 을 낸다. 같은 시드로 부르면 언제나 같은 두 수가 나오는 것은 결함이 아니라 이 모듈의 약속이다.

흔한 오해. `recv_once` 는 상대가 보낸 것을 한 번에 다 받는다

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 아홉 바이트를 보냈지만 받는 버퍼는 네 칸이다 --- 한 번에 받는 것은 버퍼만큼
  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
```

실행 결과

```
$ lowentc --run main recv_partial.low
main() = 4
```

"ping pong" 아홉 바이트를 보냈지만 네 칸 버퍼로 한 번 받으면 4 다. 받기는 버퍼만큼, 그리고 그때 도착한 만큼만 준다. 스트림에는 메시지의 경계가 없다. 보낸 쪽이 두 번에 나눠 보냈어도 한 번에 올 수 있고, 한 번에 보냈어도 나눠 올 수 있다. 메시지 단위가 필요하면 길이를 앞에 붙이거나 구분자를 정하고, 다 모일 때까지 되풀이해 받는다.

36.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`var p owned outbuf.pending be outbuf.buf_open 1 .`	표준출력으로 내보낼 대기 출력	잊으면 `E-OWN-INCOMPLETE`
`outbuf.buf_write out p buf s → result (owned pending) …`	모으고, 차면 내보내고, 새 대기 값을 준다	아는 값이 늘 하나 — 옛 값은 `E-OWN-MOVED`
`outbuf.buf_finish out p buf`	남은 바이트를 내보내고 끝낸다	완결 — 실패할 수 있다
`net.pair_of k · net.send_all · net.recv_once · net.shut_pair`	연결 쌍 · 다 보내기 · 한 번 받기 · 닫기	`cap net` 이 첫 인자 — 받기는 버퍼만큼
`random.step seed · random.bytes k dst`	재현되는 다음 상태 · 운영체제 엔트로피(`cap random`)	셈과 권위를 가른다
`clock.now_ns k · clock.since_ns k start`	단조 시계 — 경과 시간	벽시계와 약속이 다르다 — `cap clock`, 효과 `none`
`http.method_code req · http.version_ok req`	요청 줄 해석(순수)	애매한 입력을 거절한다
`aead · gcm · x25519 · ed25519 · tls13 · tlssrv`	봉인 · 키 합의 · 서명 · TLS 계산	혼자 쓰면 틀리는 조각은 문서가 경고한다

표 36.2 — 입출력·네트워크·암호 모듈의 모양 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`outbuf` 는 출력을 모았다가 내보내고, 비우지 않은 대기 값은 번역이 거절한다. `net` 의 연결은 `cap net` 으로 여는 자원이며 닫기가 실패할 수 있다. `random.step` 은 재현되는 순수 계산이고 `random.bytes` 는 `cap random` 으로 얻는 엔트로피다. 단조 시계와 벽시계는 약속이 다르다. `http` 파서는 애매한 입력을 거절하는 순수 계산이다. 암호 모듈은 유도·봉인·키 합의·서명·TLS 계산 순으로 쌓여 있고, 전송하는 TLS 는 아직 없다.

37 터미널 — `term` 과 `tty`

이 장의 차례

37.1 `term` 은 화면에 쓰지 않는다 439
37.2 바뀐 칸만 다시 그린다 440
37.3 칸의 수는 바이트의 수가 아니다 440
37.4 키를 읽는 것은 계산이다 441
37.5 raw 모드는 권한이다 442
37.6 무엇을 아직 안 지었나 443
37.7 흔한 실수 443
37.8 이 장의 문법 한눈에 445

먼저 알아야 할 것

33장 글자와 부호화 · `fmt` 는 호출자의 버퍼에 조립하고 다음 자리를 돌려준다
36장 입출력·네트워크·시간·난수·암호 · 셈과 권위를 가른다

돌아보기

36장에서 `http` 파서와 `net` 이 다른 모듈인 까닭은 무엇이었는가?

> **답.** 바이트를 받는 일(권위 — `cap net` 이 필요하다)과 그 바이트를 해석하는 일(셈 — 순수하다)이 다른 일이기 때문이다. 합치면 순수한 해석까지 권한과 효과를 끌고 다니고 시험하기 어려워진다. 이 장의 터미널 모듈도 똑같이 반으로 갈라져 있다.

이 장의 필요성과 맥락

터미널 프로그램은 두 가지를 한다. 화면에 보낼 제어 바이트를 짓고, 사용자의 키를 읽는다. 둘 다 운영체제와 닿는 부분은 아주 얇고, 나머지는 전부 계산이다. 화면을 얼마나 다시 그려야 하는지, 한글 한 글자가 몇 칸을 차지하는지, `ESC [ A` 가 무슨 키인지는 계산이다. 표준 라이브러리는 이 계산을 순수한 `term·tty` 파싱으로, 운영체제에 닿는 부분을 `cap tty` 가 필요한 세 내장 op 으로 갈랐다. 라이브러리 둘러보기의 마지막 장으로, 이 가르기가 실제로 무엇을 사는지 본다.

이 장이 끝나면

`term` 이 화면에 아무것도 쓰지 않고 호출자의 버퍼에 ANSI 제어 바이트를 조립한다는 것, 화면 diff 로 바뀐 칸만 다시 그리는 방식을 익힌다. 표시 폭과 글자 묶음(grapheme cluster)이 바이트 수와 다르다는 것을 확인한다. `tty.parse_key` 로 키를 순수하게 해석하고, raw 모드를 켜고 끄는 `cap tty` 내장 op 을 쓸 때 반드시 되돌려야 하는 이유도 보게 된다.

이 장에서 답할 질문

1 칸의 폭은 터미널마다 다르지 않은가?

37.1 `term` 은 화면에 쓰지 않는다

`term` 의 op 은 전부 `effects none` 이다. 커서를 옮기고(`goto`), 색을 바꾸고(`sgr`·`color256`), 화면을 지우는(`clear`) 제어 바이트를 **호출자의 버퍼**에 조립하고 다음 쓸 자리를 돌려줄 뿐이다. 실제로 화면에 내보내는 일은 `outbuf`(36장)가 한다.

이것을 모르면 "코드는 도는데 화면에 아무것도 안 뜬다" 는 막다른 길에 빠진다. 그러나 이 가르기 덕분에 화면 코드 전부를 화면 없이 시험할 수 있다. 조립한 바이트를 비교하면 된다.

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]

use term .

rem 화면에 쓰지 않는다 — buf 에 제어 바이트를 이어 붙이고, 쓴 바이트 수를 돌려준다
proc paint input buf mut slice u8 . . output u64 . effects state .
do
  rem 0 부터 센 2 행 5 열로 커서를 옮긴다 — 터미널은 1 부터 세므로 ESC [ 3 ; 6 H
  let p1 option u64 be term.goto buf 0 2 5 .
  guard is_some p1 . else return 0 .
  rem 굵게(1) — ESC [ 1 m. 앞 op 이 돌려준 자리에서 이어 쓴다
  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 속성을 되돌린다(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
```

실행 결과

```
$ 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]
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` 는 `buf` 의 0 번 자리부터 커서 이동 바이트를 쓰고, 다음에 쓸 자리 6 을 `option` 으로 돌려준다. 버퍼가 모자라면 아무것도 쓰지 않고 `none` 이다. 반쯤 쓴 제어 바이트는 화면을 망가뜨리기 때문이다.
- 행과 열은 **0 부터** 센다. 터미널의 약속(ANSI)은 1 부터 세므로 `goto` 가 1 을 더해 `ESC [ 3 ; 6 H`(27 91 51 59 54 72)를 쓴다. 배열 색인과 같은 기준을 쓰게 하려는 선택이다.
- `term.sgr buf 6 1` 은 굵게를 켜는 `ESC [ 1 m` 를, 마지막 `term.sgr … 0` 은 속성을 되돌리는 `ESC [ 0 m` 를 쓴다. 그 사이의 104·105 가 글자 `hi` 다.
- 모든 op 이 "다음 자리" 를 돌려주므로 앞 op 의 답을 다음 op 의 `pos` 로 잇는다. 쓴 바이트는 모두 16 개이고, VM 이 보여 주는 `buf` 가 그 바이트 그대로다.

37.2 바뀐 칸만 다시 그린다

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
```

실행 결과

```
$ lowentc --run main diffing.low
main() = 8
```

`term.diff prev next w out pos` 는 직전 화면 `prev` 와 새 화면 `next` 를 비교해 바뀐 연속 구간만 "그 자리로 가서 이것을 써라" 로 `out` 에 쓴다. 두 화면이 같으면 0 바이트다 — **0 바이트가 곧 "화면을 건드리지 마라" 는 답**이다. "hello" 가 "help!" 로 바뀌면 커서 이동과 바뀐 두 글자로 8 바이트다. 매번 화면 전체를 다시 그리면 깜빡이고 원격 접속에서는 전송량도 크다.

37.3 칸의 수는 바이트의 수가 아니다

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
```

실행 결과

```
$ 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` 는 2 바이트, 2 칸이다.
- “안녕” 은 6 바이트이지만 한글은 전각이라 4 칸이다.
- 웃는 얼굴 이모지(U+1F600)는 2 칸이다.
- `e` 다음에 결합용 악센트(U+0301)가 오고 `x` 가 오는 4 바이트는 사람이 보기에 글자 두 개(é, x)다. `row_clusters` 가 그 묶음을 센다.

터미널에서 줄을 맞추거나 잘라낼 때 바이트나 코드포인트로 세면 한글과 이모지에서 칸이 어긋난다. `row_width·fit_width·cluster_len` 이 칸과 글자 묶음으로 센다. 폭 표는 `unicode` 모듈처럼 유니코드 자료에서 뽑았다(33장).

> **문. 칸의 폭은 터미널마다 다르지 않은가?**
>
> **답.** 다르다. 동아시아의 “모호한 폭” 글자나 새 이모지를 터미널마다 다르게 그린다. `term` 은 유니코드 표준의 표를 따르고, 그 선택과 한계를 모듈 문서에 적어 둔다. 표시 폭은 틀려도 1 이라는 쓸 만한 기본값이 있어서 조금 어긋나도 화면이 무너지지는 않지만, 글자인지를 판정하는 `unicode` 의 표는 그런 기본값이 없어 기계적으로 뽑는 것이 필수였다.

37.4 키를 읽는 것은 계산이다

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
```

실행 결과

```
$ 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]
```

터미널이 방향키 위를 보내는 것은 `ESC [ A` 세 바이트다. `tty.parse_key buf 0` 은 그 바이트를 키 코드와 소비한 길이를 담은 값 하나로 읽는다. `tty.key_of` 로 키 코드를, `tty.len_of` 로 길이를 꺼낸다. 평범한 글자는 그 바이트 값(1 … 255) 그대로이고, 특수키는 1000 위에 둔다(`key_up` 은 1001). 그래서 "이건 글자다" 와 "이건 방향키다" 를 한 값으로 구별한다.

바이트열을 키로 읽는 것은 입출력이 아니라 계산이라 `effects none` 이고, 화면 없이 VM·네이티브 대조로 검증된다. 모르는 시퀀스나 모자란 바이트는 `none` 이다.

37.5 raw 모드는 권한이다

운영체제와 닿는 부분은 세 내장 op 뿐이다. `tty_raw t on`(raw 모드 들어가기·나오기), `tty_read t buf`(키 바이트 읽기), `tty_size t`(화면 크기). 모두 `cap tty` 를 첫 인자로 받는다.

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
```

실행 결과

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

- 버퍼를 raw 모드에 들어가기 **전에** 받는다. 들어간 뒤에 할당이 실패해 돌아가면 터미널이 raw 모드로 남는다.
- `tty_read` 로 읽은 바이트를 `tty.parse_key` 로 해석하고, `q` 를 누르면 끝낸다.
- 마지막에 `tty_raw t false` 로 반드시 되돌린다.

raw 모드는 **사용자의 터미널 설정을 바꾼다**. 프로그램이 죽어도 그 변경은 남아서, 사용자의 셸에서 입력이 보이지 않고 줄바꿈이 먹지 않는다. 그런 권한이 주변에 떠돌면 아무 라이브러리나 남의 셸을 망가뜨릴 수 있다. 그래서 들고 있는 자만 행사한다. 이 예제는 키 입력을 기다리므로 검증 스크립트는 검사만 한다.

> **흔한 오해. raw 모드를 되돌리는 코드도 소유로 강제하면 된다**
>
> 좋은 생각이지만 이 판에서는 그렇게 되어 있지 않다. `tty_raw` 는 불리언을 돌려주는 내장 op 이고, "raw 모드에 들어가 있음" 을 나타내는 소유 값이 없다. 그래서 되돌리기를 잊어도 번역이 막지 않는다. `outbuf` 와 `files` 가 완결을 소유로 강제하는 것과 대조되는 자리이고, 지금은 사람이 기억해야 하는 규율이다. 모듈 문서도 이 점을 굵게 경고한다.

37.6 무엇을 아직 안 지었나

`tty` 모듈 문서가 스스로 적은 목록이다 — 마우스 보고, 붙여 넣기 구분(bracketed paste), 터미널 질의 응답의 해석, 화면 크기 변경 신호(SIGWINCH) 연동. 각각 따로 지을 조각이다. 이 목록이 소스와 문서에 있으므로, 이 모듈로 편집기를 지으려는 사람은 무엇을 스스로 해야 하는지 들여오기 전에 안다.

37.7 흔한 실수

> **반례. raw 모드에 들어간 뒤 되돌리지 않고 떠나는 길을 만든다**

`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 ✘ raw 모드에 들어간 *뒤에* 할당하고, 실패하면 되돌리지 않고 떠난다
```

```
  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
```

실행 결과

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

이 코드는 번역을 통과한다(`tty_raw` 의 되돌리기는 소유로 강제되지 않는다). 그런데 할당이 실패하면 `return 2` 로 떠나면서 터미널을 raw 모드로 남긴다. 사용자의 셸에서 입력이 보이지 않고 줄바꿈이 어긋난다. 이 장의 `rawmode.low` 처럼 실패할 수 있는 준비(버퍼 받기)를 raw 모드에 들어가기 **전에** 모두 끝내고, 들어간 뒤의 모든 길이 `tty_raw t false` 를 지나게 짠다.

반례. 줄을 바이트 수로 잘라 화면 폭에 맞춘다

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 ✘ 앞 다섯 바이트로 자른다 --- 둘째 글자 한가운데서 잘려 올바른 글자열이 아니다
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 칸 수로 자를 자리를 묻는다 --- 세 칸에 들어가는 가장 긴 앞부분은 "안" 의 3 바이트
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
```

실행 결과

```
$ 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]
```

"안녕" 을 앞 다섯 바이트로 자르면 둘째 글자 한가운데서 잘려 올바른 글자열이 아니므로 `row_width` 가 `none`(999)을 준다. 화면에 그대로 내보냈다면 깨진 글자가 찍혔을 것이다. `term.fit_width row 3` 은 세 칸에 들어가는 가장 긴 앞부분을 **바이트 길이로** 알려 준다 — "안" 하나, 3 바이트. 자르기는 언제나 그 답으로 한다.

흔한 오해. 한 번 읽은 바이트에는 키 하나가 온전히 들어 있다

examples/ch37/partial_key.low

```
module partial_key .
rem run: first_key [27,91]
rem run: first_key [27,91,65]

use tty .

rem 한 번에 읽은 바이트가 키 하나를 다 담는다는 보장은 없다
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
```

실행 결과

```
$ 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]
```

방향키 위는 `ESC [ A` 세 바이트인데, 읽기가 앞 두 바이트만 가져올 수 있다. 그때 `tty.parse_key` 는 `none`(여기서 0)을 주고, 셋째 바이트까지 모이면 위(1)로 읽는다. 읽은 바이트를 버리지 말고 남은 조각을 다음 읽기 앞에 이어 붙인다. `none` 을 "모르는 키" 로 버리면 방향키가 가끔 사라지는 결함이 된다.

37.8 이 장의 문법 한눈에

모양	뜻	왜 이렇게
`term.goto` · `term.sgr` · `term.clear` `(buf pos …)`	제어 바이트를 호출자의 버퍼에 조립	화면에 쓰지 않는다 — 바이트로 시험한다
`term.diff prev next w out pos`	바뀐 구간만 다시 그리는 바이트 — 같으면 0	깜빡임과 전송량을 줄인다
`term.row_width row` · `term.row_clusters row` · `term.cp_width cp`	칸 수 · 글자 묶음 수 · 코드포인트 폭	칸은 바이트가 아니다

`term.fit_width row cols`	칸에 들어가는 가장 긴 앞부분의 바이트 길이	글자 한가운데서 자르지 않는다
`tty.parse_key buf at • tty.key_of • tty.len_of`	키 바이트 해석(순수) — 모자라면 `none`	키 읽기는 계산이다
`tty_raw t true • tty_read t buf • tty_size t`	raw 모드 · 키 바이트 읽기 · 화면 크기	`cap tty` — raw 모드는 반드시 되돌린다
`term.goto buf 0 2 5`	0 부터 센 행·열 — 바이트는 `ESC [ 3 ; 6 H`	배열 색인과 같은 기준, 터미널의 1 기준은 op 이 맞춘다

표 37.1 — 터미널 모듈의 모양 — 모양 · 뜻 · 왜 이렇게 생겼나

복습 정리

`term` 은 ANSI 제어 바이트를 호출자의 버퍼에 조립하는 순수 모듈이고, 화면에 내보내는 일은 `outbuf` 가 한다. `diff` 는 바뀐 칸만 다시 그리며 같으면 0 바이트다. 칸의 수는 바이트 수와 다르므로 `row_width·row_clusters` 로 센다. `tty.parse_key` 는 키 바이트를 순수하게 해석한다. 운영체제와 닿는 `tty_raw·tty_read·tty_size` 는 `cap tty` 를 받고, raw 모드는 반드시 되돌린다.

제10부 — 근거: 무엇이 증명되었나

이 부는 이 언어가 스스로에 대해 하는 주장이 **어디까지 참인지**를 다룬다. 앞의 장들에서 "이것은 증명되어 있다" 는 말이 여러 번 나왔다. 이 부는 그 말들이 각각 무엇을 뜻하고, 어떤 모델에 대한 것이며, 어디서 끝나는지를 정직하게 모은다.

왜와 도구 (38·39장) — 시험은 있음을, 증명은 없음을 보인다. 증명·전수 검사·대조를 겹쳐 쓰는 이유와 증명된 것의 규모, 그리고 뒤의 장이 쓰는 수학 여섯 가지(부분 순서·격자·고정점·귀납·추상해석)를 처음부터 모은다.

수와 메모리 (40–43장) — 넓히기와 나눗셈, 경계 검사를 지우는 구간·관계·행우선 증명, 소유와 차용의 중심 정리, 몇 번을 돌든 안전한 되풀이.

효과와 동시성 (44–47장) — 효과 선언이 실제를 덮는다는 정리, 규율이 경합을 없애고 병렬이 순차와 같다는 정리, 약한 메모리 모델, 그리고 무거운 도구가 실제로 필요한 락.

표면과 정체 (48·49장) — 어느 닫개로 닫아도 같은 나무가 나온다는 문법의 증명과, 같은 프로그램을 뜻으로 알아보는 해시.

무엇이 증명되지 않았나 (50장) — 신뢰 기반, 모델과 구현 사이의 간극, 주제별로 남은 것. 이 부의 마지막 장이 본문의 마지막 장인 것은 편집상의 우연이 아니다. 그 장을 읽지 않고 이 부를 인용하면, 이 매뉴얼이 말하지 않은 것을 말한 것으로 만들게 된다.

Coq 을 읽지 않아도 되도록 정리를 평서문으로 옮기고, 각 정리가 컴파일러의 어떤 거절로 이어지는지 예제로 보인다. 필요한 수학은 고등학교 수준이고 39장에서 처음부터 설명한다. 건너뛰어도 본문이 이어지게 쓴 형식적 설명은 수학 상자에 담았다.

38 왜 증명하는가

이 장의 차례

38.1 시험은 있음을, 증명은 없음을 448
38.2 세 층을 겹친다 450
38.3 무엇이 증명되어 있나 450
38.4 증명하는 세 가지 방법 451
38.5 이 부를 읽는 약속 452

먼저 알아야 할 것

31장 짓고 시험하기 · 시험·두 백엔드 대조·계약 대조
14장 계약 · 강제되는 계약은 사실이 되어 검사를 지운다

돌아보기

31장에서 두 백엔드 대조 같은 검사는 결함의 **무엇**을 보이고 **무엇**은 보이지 못한다고 했는가?

> **답.** 결함이 **있음**은 보이지만 **없음**은 보이지 못한다고 했다. 두 구현이 같은 답을 낸다고 둘 다 맞는 것은 아니기 때문이다. 그리고 없음은 증명의 몫이라고 했다. 제10부가 그 증명을 다룬다.

이 장의 필요성과 맥락

제10부는 이 언어가 스스로에 대해 하는 주장이 어디까지 참인지를 다룬다. 앞의 장들에서 "이것은 증명되어 있다" 는 말이 여러 번 나왔다 — 넓히기, `mod` 의 색인 안전, 차용 규칙, 병렬의 결정성. 이 부는 그 말들이 각각 무엇을 뜻하고, 어떤 모델에 대한 것이며, 어디서 끝나는지를 정직하게 모은다. 그 첫 장은 왜 증명이 필요한지, 그리고 증명·전수 검사·대조라는 세 층이 어떻게 겹치는지를 세운다.

이 장이 끝나면

시험이 "있음(∃)" 을, 증명이 "없음(∀)" 을 보인다는 구별을 예제로 확인한다. 증명·전수 검사·대조 세 층이 각각 무엇을 잡고 무엇을 놓치는지, 이 저장소의 증명이 쓰는 세 전략(불변식 보존·순서 구조로 환원·전수 검사)을 알게 된다. 증명된 것의 실측 규모와, 증명이 컴파일러가 무엇을 거절해야 하는지를 정한다는 관계도 이해하게 된다.

이 장에서 답할 질문

1 증명이 컴파일러 코드와 어떻게 이어지는가? 정리가 있다고 컴파일러가 달라지는가?

38.1 시험은 있음을, 증명은 없음을

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
```

실행 결과

```
$ 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 ==
```

두 수의 평균을 구하는 `mean2` 는 시험 세 개를 모두 통과한다. 그런데 4294967295 와 1 을 주면 `add a b` 가 `u32` 에서 넘쳐 멈춘다. 시험을 백만 개 더 써도 저자가 큰 수를 고르지 않으면 이 결함은 드러나지 않는다.

이것이 시험의 본성이다. 시험은 "**이** 입력에서 틀린다(∃)" 를 보일 수 있지만 "**어떤** 입력에서도 틀리지 않는다(∀)" 를 보일 수 없다. 자연수를 백만 개 확인해도 모든 자연수를 확인한 것이 아니다. ∃ 를 아무리 모아도 ∀ 가 되지 않는다. 이 간극이 증명이 필요한 유일한 이유다.

```
mean2 의 입력: u32 두 개 = 약 1.8 × 10^19 쌍

시험   (10,20) (0,0) (10^6,3·10^6)   세 점을 찍었다 → 셋 다 맞다
반례   (4294967295, 1)               찍지 않은 자리 → 넘쳐 멈춘다
증명   ── 모든 쌍 ──────────        한 번에 모두에 대해 말한다
```

Lowent 에서 이 결함은 적어도 **조용하지는 않다**. C 였다면 넘친 합이 감겨 틀린 평균을 냈을 것이다. 여기서는 멈춘다(4장). 그리고 고친 판은 넘칠 수 없는 모양으로 적는다.

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
```

실행 결과

```
$ 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%)
```

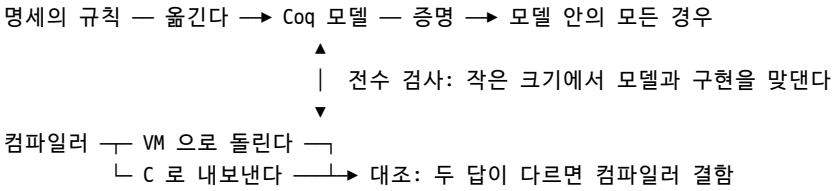
작은 쪽에 두 수 차이의 절반을 더하면 넘칠 수 없다. `--ir` 은 세 검사 가운데 하나를 지웠다고 말한다. 남은 두 검사가 실제로 넘칠 수 없는데도 남는 것은 구간 분석이 `min·max` 의 관계를 모르기 때문이다. 분석은 **안전한 방향**으로만 틀린다 — 증명하지 못하면 검사를 남긴다.

38.2 세 층을 겹친다

층	무엇을 하나	얼마나 센가	무엇을 놓치나
증명	모든 경우를 논리로	모델 안에서 반례가 존재할 수 없다	모델이 틀리면 무력하다
전수 검사	정해진 크기의 모든 경우를 실제로	그 크기 안에서 확실하다	큰 입력을 못 본다
대조	두 구현을 같은 입력으로 맞대 본다	불일치를 잡는다	둘이 똑같이 틀리면 못 잡는다

표 38.1 — 검증의 세 층

셋이 다른 종류의 실수를 잡기 때문에 겹쳐 쓴다. 누가 무엇을 맞대는지 그리면 이렇다.



증명은 모델을, 대조는 구현을 보고, 전수 검사는 그 둘 사이를 잇는다. 이 저장소에서는 증명이 못 잡은 결함을 대조가 잡았고, 대조가 못 잡은 결함을 증명이 잡았다. 그리고 이 부는 세 낱말을 섞지 않는다. "증명됨" 은 Coq 이 모든 경우를 확인했다는 뜻이고, "전수 검사됨" 은 정해진 크기의 모든 경우를 돌렸다는 뜻이며, "스케치" 는 사람이 논증했을 뿐 기계가 확인하지 않았다는 뜻이다.

> **흔한 오해. 증명된 언어로 짠 프로그램은 옳다**
>
> 증명은 **언어의 규칙**에 대한 것이지 **당신의 프로그램**에 대한 것이 아니다. 차용 규칙이 건전하다는 정리는 "번역을 통과한 프로그램에 차용 위반이 없다" 를 말할 뿐, 그 프로그램이 원하는 답을 낸다는 것을 말하지 않는다. 위의 `mean2` 는 차용 위반도 없고 넘침도 조용하지 않지만, 여전히 틀린 설계였다. 프로그램이 옳은지는 계약·시험·검토가 답한다.

38.3 무엇이 증명되어 있나

증명 파일은 모두 `docs/proofs/coq/` 에 있다. 이 판에서 센 규모다.

항목	수
Coq 파일	31(검사기 추출 파일 1 포함)

정리(Theorem) · 따름정리(Corollary) · 보조정리(Lemma)	179 · 15 · 250
예(Example)	33
`Qed`	481
`Admitted` · `Axiom`	0
증명 스크립트 줄 수	10,350
검증 도구	Rocq 9.2(전부) · Coq 8.20.1(약한 메모리 둘을 뺀 나머지)

표 38.2 — 기계 검증된 증명의 규모

Coq 에서 `Admitted` 는 "이건 참이라고 치고 넘어간다" 이고 `Axiom` 은 "이건 가정한다" 다. 그런 자리가 하나라도 있으면 그 파일의 모든 정리가 그 가정에 기댄다. 둘 다 0 이다. 무엇을 증명했고 무엇을 아직 안 했는지는 `docs/proofs/LEDGER.md` 에 적혀 있다.

묶음으로 보면 이렇다.

묶음	한 문장으로	장
도구	뒤의 장이 쓰는 수학 여섯 가지 — 부분 순서·격자·고정점·귀납·추상해석	39장
수	값이 조용히 변하는 넓히기는 없고, 나눗셈이 실패하는 경우는 정확히 알려져 있다	40장
경계	검사를 지운 자리는 범위 안임이 증명된 자리다	41장
소유와 차용	번역을 통과한 프로그램은 실행 중 차용 위반이 없다	42장
되풀이	고정점에 닿는 반복은 몇 번을 돌든 차용 위반이 없다	43장
효과	효과 선언은 실제로 일어나는 효과를 덮고, 그 위에서 최적화가 적법하다	44장
경합과 병렬	안전 코드에는 데이터 경합이 없고, 병렬 결과는 순차와 비트까지 같다	45장
약한 메모리	기본 기억 차례면 순서대로 생각해도 되고, 약하게 적으면 거동이 늘어난다	46장
락	CAS 스핀락은 상호배제를 주고, 오름차순 잠금은 교착하지 않는다	47장
문법	한 문장은 한 문장짜리 블록이고, 어느 닫개로 닫아도 같은 나무가 나온다	48장
해시	주석은 해시를 바꾸지 않고, 인코딩이 같으면 인터페이스가 같다	49장

표 38.3 — 증명된 것의 묶음과 이 부의 장

마지막 장(50장)은 이 모두가 기대는 것과 덮지 않는 것을 모은다.

38.4 증명하는 세 가지 방법

증명 스크립트를 읽을 필요는 없다. 그러나 **어떤 방식으로** 증명했는지 알면 그 정리를 얼마나 믿어야 하는지 감이 온다.

불변식 보존. "프로그램이 한 걸음 갈 때마다 어떤 성질이 유지된다" 를 보인다. 시작에서 참이고(기초) 한 걸음이 참을 참으로 넘기면(보존), 몇 걸음을 가도 참이다. 수학적 귀납법의 프로그램 판이다. 차용 규칙과 반복의 정리가 이 방식이다.

순서 구조로 환원. 문제를 어떤 순서에 관한 문제로 바꾼 뒤 순서의 성질로 답한다. "`u8` 은 `u32` 에 안전하게 들어간다" 를 부분 순서로 만들면, "두 타입을 합칠 때 무엇을 골라야 하나" 가 최소 상계라는 잘 알려진 개념이 된다. 수의 넓히기와 약한 메모리 모델이 이 방식이다.

전수 검사. 아직 모든 크기에 대해 증명하지 못한 성질은 크기를 정해 그 안의 모든 경우를 돌린다. 이것은 증명이 아니고, 이 부는 그렇게 적는다.

38.5 이 부를 읽는 약속

이 부는 세 가지를 지킨다.

- **낱말을 섞지 않는다.** “증명됨”·“전수 검사됨”·“스케치” 는 서로 다른 세기다. “이 언어는 증명됐다” 가 사실은 “짧은 경우들을 돌려 봤다” 인 경우가 흔하다.
- **정리마다 막는 결함이 붙는다.** 무엇을 막는지 적을 수 없는 정리는 싣지 않았다. 읽는 사람에게 값이 없는 정리는 여기서 다룰 까닭이 없다.
- **이 부는 규범이 아니다.** 언어의 정의는 명세의 조항 정본이다. 이 부와 명세가 어긋나면 이 부가 틀렸다.

그리고 이 언어가 기대는 생각을 여섯 줄로 줄이면 이렇다. 여섯 모두 수학으로 뒷받침되거나, 뒷받침되지 않는다고 적혀 있다 — 그 구별이 이 부의 목적이다.

1. **틀린 프로그램은 쓸 수 없게 만든다.** 틀린 것을 잡는 것보다 표현할 수 없게 하는 쪽이 싸다(42장).
2. **모르는 것은 모른다고 답한다.** 실패는 `option`·`result` 라는 값이지 숨은 예외가 아니다.
3. **비용은 보여야 한다.** 할당·곁효과·복사가 모두 소스에 적힌다(44장).
4. **의심스러우면 검사를 남긴다.** 검사를 지우는 것은 증명이 있을 때뿐이다(41장).
5. **두 구현이 같은 답을 내야 한다.** VM 과 네이티브가 다르면 컴파일러 결함이다(31장).
6. **못 한 것은 적는다.** 가정·한계·미증명을 남긴다(50장).

> **문. 증명이 컴파일러 코드와 어떻게 이어지는가? 정리가 있다고 컴파일러가 달라지는가?**
>
> **답.** 정리는 **컴파일러가 무엇을 거절해야 하는지**를 정한다. 넓히기 표에서 `uN ⊑ iM` 의 조건이 `N < M`(엄격)이라는 것, 루프를 넘어 사는 쓰기 빌림과 본문의 소유자 쓰기가 만나는 모양 **하나만** 막으면 충분하다는 것이 정리에서 나왔다. 반대 방향도 있다. 구현이 겪은 결함(“차용을 함수로 세탁하기”)을 정리로 옮겨 두면, 누가 그 검사를 지우는 순간 증명이 깨진다. 이야기는 잊히지만 정리는 남는다(42장).

```
정리 ──────────→ 컴파일러가 거절할 모양을 정한다
                 예: uN ⊑ iM 의 조건은 N < M (엄격)
구현이 겪은 결함 → 정리로 옮긴다
                 예: 차용을 함수로 세탁하기
                 그 검사를 지우면 증명이 깨진다
```

> **복습 정리**
>
> 시험은 틀리는 입력이 **있음**을 보이고, 증명은 모든 입력에서 틀리지 **않음**을 보인다. Lowent 는 증명·전수 검사·대조를 겹쳐 쓰고, “증명됨”· “전수 검사됨”·“스케치” 를 섞지 않는다. 31 개 Coq 파일에 `Admitted`·`Axiom` 이 없다. 증명은 불변식 보존·순서 구조 환원·전수 검사로 이루어지며, 컴파일러가 무엇을 거절해야 하는지를 정한다. 증명은 언어의 규칙에 대한 것이지 당신의 프로그램이 옳다는 보증이 아니다.

39 수학 도구상자

이 장의 차례

39.1 여섯 가지 도구 454
39.2 집합과 관계 454
39.3 부분 순서 — 비교할 수 없는 쌍 454
39.4 격자와 join — 합칠 때 무엇을 고르나 454
39.5 단조 함수와 고정점 — 분석은 왜 멈추는가 455
39.6 귀납법과 불변식 — 도미노 456
39.7 추상해석 — 값 대신 범위로 456
39.8 기호 표 457
39.9 도구들이 맞물리는 방식 457

먼저 알아야 할 것

38장 왜 증명하는가 · 시험은 있음(∃)을, 증명은 없음(∀)을 보인다
4장 수 · 폭과 부호가 타입을 정하고, 넓히기는 값을 지킬 때만 저절로 일어난다
7장 흐름 · `while` 은 조건이 참인 동안 몸을 되풀이한다

돌아보기

38장에서 증명하는 세 가지 방법은 무엇이었는가?

> **답.** 불변식 보존(한 걸음이 참을 참으로 넘긴다), 순서 구조로 환원(문제를 순서의 성질로 바꾼다), 전수 검사(정해진 크기의 모든 경우를 돌린다 — 증명은 아니다)였다. 앞의 둘에는 수학의 어휘가 필요하다. 이 장이 그 어휘를 처음부터 모은다.

이 장의 필요성과 맥락

제10부의 나머지 장은 부분 순서, 격자, 고정점, 귀납, 추상해석이라는 낱말을 설명 없이 쓴다. 그 낱말들은 어려워 보이지만 뜻은 대부분 고등학교 수학이다 — 집합, 부등식, 도미노처럼 넘어가는 귀납. 이 장은 여섯 가지 도구를 Lowent 의 실제 프로그램에 붙여 한 번씩 보여 준다. 여기서 배운 기호는 뒤의 모든 장에서 그대로 쓰인다. 무엇을 증명한 장이 아니라 읽을 준비를 하는 장이다.

이 장이 끝나면

집합과 관계, 비교할 수 없는 쌍을 허락하는 부분 순서와 그 세 조건, 두 값을 합칠 때 고르는 join 과 격자를 알게 된다. 단조 함수와 고정점이 반복 분석을 끝나게 한다는 것, 귀납법과 불변식이 "몇 걸음이든" 을 보인다는 것, 값 대신 범위로 계산하는 추상해석과 그 건전성의 방향을 익힌다. 기호 표와, 여섯 도구가 뒤의 장에서 어떻게 맞물리는지도 보게 된다.

이 장에서 답할 질문

1 이 장에서 증명된 것은 무엇인가?

39.1 여섯 가지 도구

도구	한 줄	어디서 쓰나
집합과 관계	무엇들이 있고, 무엇이 무엇과 짝인가	전부
부분 순서	크기 비교인데 비교할 수 없는 쌍이 허락되는 순서	40·46장
격자와 join	두 개를 합칠 때 둘을 다 담는 가장 작은 것	40·44장
단조 함수와 고정점	더 이상 변하지 않는 점 — 반복 분석이 여기서 멈춘다	41·43장
귀납법과 불변식	첫 조각과 넘기는 규칙이 있으면 모든 조각이 넘어간다	42·43장
추상해석	값 대신 값의 범위로 계산한다	41장

표 39.1 — 뒤의 장에서 쓰는 수학

39.2 집합과 관계

집합은 모음이다 — `{0, 1, 2}`. **관계**는 짝의 목록이다. "작다" 관계는 `{(0,1), (0,2), (1,2)}` 처럼 순서쌍의 모음이다. 프로그램 이야기에서는 타입의 집합 `{u8, u16, u32, u64, i8, …}` 위에 "안전하게 넓혀진다" 관계를 둔다. 그 관계를 `⊑` 로 적고 "왼쪽이 오른쪽에 값이 변하지 않고 들어간다" 로 읽는다. 부등호 `≤` 와 닮았지만 수의 크기가 아니라 **담을 수 있는가**를 말한다.

39.3 부분 순서 — 비교할 수 없는 쌍

수는 언제나 비교된다. 3 과 5 가운데 하나는 크다. 타입은 그렇지 않다. `u8 ⊑ u16 ⊑ u32 ⊑ u64` 이고 `i8 ⊑ i16 ⊑ i32 ⊑ i64` 이지만, `u32` 와 `i32` 는 어느 쪽도 상대를 다 담지 못한다. `u32` 는 40억까지 담고 `i32` 는 음수를 담는다. 어느 쪽이 크다고 말할 수 없다는 사실을 정직하게 담는 구조가 부분 순서다.

조건	뜻	예
반사성	자기 자신에는 늘 들어간다	`u8 ⊑ u8`
추이성	징검다리로 이어진다	`u8 ⊑ u16` 이고 `u16 ⊑ u32` 이면 `u8 ⊑ u32`
반대칭성	서로 들어가면 같은 것이다	`A ⊑ B` 이고 `B ⊑ A` 이면 `A = B`

표 39.2 — 부분 순서의 세 조건

이 셋이 **증명되어야** "안전한 넓히기" 라는 말이 뜻을 갖는다. 추이성이 깨지면 `u8 → u16` 과 `u16 → u32` 는 허락하면서 `u8 → u32` 는 막는 언어가 된다. 사용자는 왜 안 되는지 영영 알 수 없다. 이 셋은 40장에서 실제 타입에 대해 증명된다.

39.4 격자와 join — 합칠 때 무엇을 고르나

두 값을 한 연산에 넣으면 처리기는 **둘을 다 담는 타입**에서 계산해야 한다. 후보는 여럿이지만 가장 작은 것을 고르는 것이 맞다. 그 "가장 작은 공통 상위" 를 **join** 이라 하고 `⊔` 로 적는다. 임의의 두 원소가 늘 join 을 갖는 부분 순서를 **격자**라 한다.

examples/ch39/wider.low

```
module widening .
rem run: wider 200 60000
rem trap: wider 255 65535
```

```
rem u8 과 u16 을 더하면 둘을 다 담는 가장 작은 타입(u16)에서 계산된다
fn wider input a u8 . input b u16 . output u32 . do
  return add a b .
end
```

실행 결과

```
$ 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` 이다. 그래서 출력 타입이 `u32` 여도 덧셈은 `u16` 에서 일어나고, `255 + 65535` 는 `u16` 을 넘쳐 멈춘다. join 은 **피연산자**에서 정해지지, 결과를 받을 자리에서 정해지지 않는다.

부호가 섞이면 직관과 달라진다. `u8 ⊔ i8` 은 `i8` 이 아니라 `i16` 이다 — `u8` 의 255 와 `i8` 의 −128 을 함께 담으려면 폭이 더 넓어야 한다. 사람은 이런 자리에서 틀린다. Lowent 는 이 넓히기를 대신 고르지 않고 적게 한다.

examples/ch39/sign_join.low

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

rem u8 과 i8 을 둘 다 담는 타입은 i16 이다. 처리기는 그 넓히기를 대신 고르지 않는다
fn mix input a u8 . input b i8 . output i16 . do
  return add a b .
end
```

실행 결과

```
$ 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)
```

진단이 말하는 “strictly wider signed type” 이 바로 `u8 ⊔ i8 = i16` 이다. 둘을 모두 `i16` 으로 넓혀 적으면 받아들여진다.

39.5 단조 함수와 고정점 — 분석은 왜 멈추는가

단조 함수는 입력이 커지면 출력이 작아지지 않는 함수다. **고정점**은 `f(x) = x` 인 점, 곧 한 번 더 적용해도 변하지 않는 점이다.

처리기가 반복을 분석할 때 이 두 개념이 일한다. 분석기는 반복에 들어올 때 변수의 범위를 모른다. 그래서 좁게 시작해 되풀이하며 넓힌다.

```
1 바퀴 뒤: i ∈ [0, 0]
2 바퀴 뒤: i ∈ [0, 1]
3 바퀴 뒤: i ∈ [0, 2]      … 이대로면 끝나지 않는다
```

그래서 **위드닝**(widening)을 쓴다. 몇 번 넓어지는 것을 보면 "계속 커지겠다" 고 보고 한 번에 위 끝까지 민다. `i ∈ [0, ∞)` 에 반복 조건 `i < 10` 을 합치면 `i ∈ [0, 9]` 이고, 한 바퀴 더 돌려도 `[0, 9]` 다. **고정점에 닿았고** 분석이 끝난다.

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
```

실행 결과

```
$ 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
```

증명 줄이 하나다. `add i 1` 은 `i ∈ [0, 9]` 에서 더하므로 `u8` 을 넘지 않는다고 증명되었다(`R-ARITH-RANGE 0 9`). `add s i` 는 증명되지 않았다. 위드닝이 `s` 의 위 끝을 버렸기 때문이다. 실제로 `s` 는 45 를 넘지 않지만 분석은 그것을 모른다. 그래서 검사가 남는다 — 느릴 뿐 틀리지는 않는다.

- 고정점이 늘 있다는 것이 분석이 **멈춘다**는 보장이다.
- 위드닝은 **넓게** 잡는다. 실제보다 크게 잡으면 검사를 못 지울 뿐 틀린 답은 나오지 않는다. 이 방향이 추상해석의 핵심이다.

39.6 귀납법과 불변식 — 도미노

수학적 귀납법은 이렇다. `P(0)` 이 참이고, `P(n)` 이 참이면 `P(n+1)` 도 참이다. 그러면 모든 `n` 에 대해 `P(n)` 이 참이다. 프로그램에서는 `n` 이 "실행한 걸음 수" 가 되고, `P` 를 **불변식** — 실행 내내 참인 성질 — 이라 부른다.

> **수학. 소유와 차용 정리의 모양**
>
> 시작할 때 빌림 상태는 깨끗하다(`P(0)`). 정적 검사를 통과한 사건 하나를 실행하면 깨끗한 상태는 여전히 깨끗하다(`P(n) → P(n+1)`). 따라서 몇 걸음을 실행해도 차용 위반이 없다(`∀n, P(n)`). 이 논증의 좋은 점은 **몇 걸음인지 몰라도 된다**는 것이다. 그래서 반복을 몇 번 도는지 몰라도 결론이 선다(42·43장).

39.7 추상해석 — 값 대신 범위로

프로그램을 돌리지 않고 값에 대해 무언가 알아내려면 값 하나 대신 **값의 집합**을 다뤄야 한다. 집합을 그대로 다루면 너무 크니 간단한 근사를 쓴다. 그 대표가 구간 `[lo, hi]` 다.

```
x ∈ [0, 5],  y ∈ [10, 20]  이면
x + y ∈ [10, 25]      양 끝끼리 더한다
x × y ∈ [0, 100]      부호가 섞이면 네 곱의 최소·최대를 본다
```

규칙은 하나다. **추상 계산의 결과는 실제로 가능한 모든 값을 포함해야 한다.** 실제로 `x` 가 늘 3 인데 분석이 `[0, 5]` 라고 답하면 손해는 있어도 안전하다(검사를 못 지운다). 실제로 `x` 가 7 일 수 있는데 `[0, 5]` 라고 답하면, 그 답으로 검사를 지운 순간 범위 밖 접근이 일어난다. 그래서 분석의 모든 규칙은 "넓게" 쪽으로 기울어 있다.

> **흔한 오해. 분석이 정확할수록 좋다**
>
> 정확함보다 먼저 오는 것이 **방향**이다. 넓게 틀리는 분석은 느릴 뿐이지만 좁게 틀리는 분석은 틀린 메모리를 읽는다. 정밀도를 높이는 규칙을 더할 때마다 물어야 하는 것은 "더 정확한가" 가 아니라 "실제보다 좁게 잡는 경우가 생기는가" 다. 그래서 이 언어는 무거운 분석 대신 넓게 틀리는 가벼운 분석을 두고, 모자란 자리에서는 검사를 남긴다.

39.8 기호 표

기호	읽는 법	뜻
`A ⊑ B`	A 는 B 에 안전하게 들어간다	안전한 넓히기 관계
`A ⊔ B`	A 와 B 의 join	둘을 다 담는 가장 작은 것
`[lo, hi]`	구간	lo 이상 hi 이하의 값
`∀x, P(x)`	모든 x 에 대해 P	예외가 없다
`∃x, P(x)`	어떤 x 가 있어 P	하나라도 있다
`P → Q`	P 이면 Q	함의
`⊤`	모름	아무 값이나 가능
`A * B`	A 와 B, 서로 다른 조각	분리논리의 접속사(47장)

표 39.3 — 이 부의 기호

39.9 도구들이 맞물리는 방식

```
부분 순서 ⊑          "안전하게 담긴다" 를 정의한다      (수)
   │
   ├─ join ⊔         합칠 때 타입을 고른다              (수 · 효과)
   │
격자 위의 단조 함수   반복 분석이 그 위를 오른다         (경계)
   │
고정점               분석이 멈추는 자리                 (경계 · 되풀이)
   │
추상해석의 건전성     "넓게" 틀리면 결론이 안전하다      (경계)
   │
불변식과 귀납법       실행 내내 유지됨을 보인다          (소유 · 되풀이 · 경합)
```

아래에서 위로 읽어도 된다. 귀납법으로 불변식을 지키고, 그 불변식이 추상해석의 건전성에서 오고, 추상해석은 격자 위에서 돌고, 격자는 부분 순서 위에 선다. 도구가 없으면 설계가 어림짐작이 된다 — join 을 손으로 적은 표는 `u8 ⊔ i8 = i8` 같은 줄을 섞어 255 를 −1 로 바꾸고, 위드닝 없는 반복 분석은 끝나지 않으며, 불변식 없는 차용 검사는 "이때는 되고 저때는 안 되는" 규칙 더미가 된다.

문. 이 장에서 증명된 것은 무엇인가?

답. 없다. 이 장은 어휘를 소개했다. 반사성·추이성·반대칭성, join 의 건전성, 고정점의 존재가 Lowent 의 **실제 정의에 대해** 성립한다는 것이 40장부터의 내용이다. 그리고 추상해석의 건전성은 규칙마다 "왜 이것이 넓은가" 를 따로 확인해야 한다. 한 규칙이라도 좁게 잡으면 그 자리에서 샌다. 그래서 41장은 규칙 하나하나를 그렇게 따진다.

복습 정리

부분 순서는 비교할 수 없는 쌍을 허락하는 순서이고 반사·추이·반대칭을 지킨다. join 은 둘을 다 담는 가장 작은 것이며 피연산자에서 정해진다 (`u8 ⊔ u16 = u16`, `u8 ⊔ i8 = i16`). 반복 분석은 위드닝으로 고정점에 닿아 멈추고, 위드닝은 넓게 잡아 검사를 남길 뿐 틀리지 않는다. 귀납법과 불변식은 "몇 걸음이든" 을 보이고, 추상해석은 값 대신 범위로 계산하되 늘 실제를 포함하는 쪽으로 틀린다.

40 수의 증명 — 넓히기, 좁히기, 나눗셈

이 장의 차례

40.1 넓히기는 부분 순서다 459
40.2 나눗셈과 나머지 461
40.3 range 가 주는 정리 462
40.4 멈춤을 정직하게 모델에 넣는다 463
40.5 증명하지 않은 것 463

먼저 알아야 할 것

4장 수 · 넓히기는 저절로, 좁히기는 적는다
13장 이름 붙인 타입 · `range lo hi` 는 매개변수의 생김새가 된 계약이다
39장 수학 도구상자 · 부분 순서의 세 조건과 join

돌아보기

4장에서 `u8` 과 `i16` 은 섞을 수 있지만 `u32` 와 `i32` 는 섞을 수 없었다. 무엇이 둘을 갈랐는가?

> **답.** 값을 지키는 넓히기가 있느냐였다. `u8` 의 모든 값(0 … 255)은 `i16` 에 들어가지만, `u32` 의 큰 값은 폭이 같은 `i32` 에 들어가지 않는다. 이 장은 그 "값을 지키는 넓히기" 를 39장의 부분 순서로 정의하고, 그것이 정말 안전하다는 기계 증명을 읽는다.

이 장의 필요성과 맥락

C 에서 `unsigned char small = 300;` 은 경고 없이 44 가 되고, `-1` 을 `unsigned int` 에 넣으면 4294967295 가 된다. Lowent 는 값이 보존되는 넓히기만 저절로 허락하고, 잃어도 괜찮으면 어떻게 잃을지 골라 적게 한다(`narrow_wrap`·`narrow_sat`·`narrow_try`). 그 규칙이 옳다는 말은 표 몇 줄이 틀리지 않았다는 뜻인데, 사람은 그런 표를 틀린다. 그래서 정의를 수학으로 세우고 경우를 하나도 빠뜨리지 않았음을 기계에 맡겼다. 코드와 가장 직접 닿는 증명이라 제10부의 도구 다음에 둔다.

이 장이 끝나면

넓히기 관계 ⊑ 의 네 줄 정의와 그것이 값을 보존하는 부분 순서라는 정리, join 과 좁히기의 성질을 알게 된다. 나눗셈이 실패하는 경우가 정확히 무엇인지, `mod` 와 부호의 약속, `range` 가 검사를 지우는 근거가 되는 정리들을 익힌다. 타입 규칙의 세 정리(`progress`·`preservation`· `values_fit`)가 멈춤을 어떻게 정직하게 모델에 넣었는지도 보게 된다.

이 장에서 답할 질문

1 넓히기가 실패할 수 없다는 것은 따로 증명할 만한가?

40.1 넓히기는 부분 순서다

`t ⊑ u` 는 "타입 `t` 의 어떤 값이든 타입 `u` 에 값이 변하지 않고 들어간다" 로 읽는다. 정의는 네 줄이다.

왼쪽 → 오른쪽	조건	왜
`uN` → `uM`	$N \le M$	부호 없음끼리는 폭만 넓으면 된다
`iN` → `iM`	$N \le M$	부호 있음끼리도 같다
`uN` → `iM`	$\mathbf{N < M}$	`u8`(255)은 `i8`(127)에 안 들어간다. 폭이 **더** 커야 한다
`iN` → `uM`	없음	음수를 담을 곳이 없다

표 40.1 — 넓히기 관계 ⊑

셋째 줄의 `<` 가 요점이다. `≤` 로 적었다면 `u8 ⊑ i8` 이 허용되고 255 가 −1 로 바뀐다. 이 한 글자가 값의 안전을 가르고, 사람이 이런 것을 틀린다.

```
u8 ──▸ u16 ──▸ u32 ──▸ u64
  \      \      \
   ▾      ▾      ▾
i8 ──▸ i16 ──▸ i32 ──▸ i64

→ 는 ⊑ (값이 안 바뀌고 들어간다). 화살을 이어 가면 그것도 ⊑ 다(추이성).
u8 → i8 · u16 → i16 … 처럼 같은 폭으로 내려가는 화살은 없다 (N < M).
i 줄에서 u 줄로 올라가는 화살도 없다 (음수를 담을 곳이 없다).
```

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
```

실행 결과

```
$ 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
```

실행 결과

```
$ lowentc --run wider order_ok.low 200
wider(200) = 200
```

`NumericLattice.v` 가 이 관계에 대해 증명한 것들이다.

- **값을 보존한다**(`sub_preserves`). `t ⊑ u` 이고 `v` 가 `t` 의 범위에 있으면 `u` 의 범위에도 있다. 넓히기에 실행 중 검사가 없는 근거다.

- **부분 순서다**(`sub_refl`·`sub_trans`·`sub_antisym`). 추이성이 깨지면 `u8 → u16` 과 `u16 → u32` 는 되는데 `u8 → u32` 는 안 되는 언어가 된다. 표를 손으로 관리하면 실제로 그런 구멍이 생긴다.
- **join 은 둘 다를 담고, 둘 가운데 하나다**(`join_sound`·`join_is_an_operand`). 두 값을 한 연산에 넣을 때 고르는 타입은 두 값을 모두 담고, 소스에 없는 제3의 타입을 지어내지 않는다(39장의 `u8 ⊔ u16 = u16`).
- **좁히기의 조건은 정확히 하나다**(`narrow_ok_iff`). `narrow_try u8 300` 이 조용히 성공해 44 를 주는 일이 없다.

> **수학. 넓히기가 값을 보존한다는 증명의 뼈대**
>
> 경우를 넷으로 나눈다. `uN ⊑ uM`($N \le M$)이면 두 아래 끝이 모두 0 이고, 위 끝은 $2^N - 1 \le 2^M - 1$ 이다 — 2 의 거듭제곱이 단조라는 사실만 쓴다. `uN ⊑ iM`($N < M$)이면 $N \le M - 1$ 이므로 $2^N - 1 \le 2^{M-1} - 1$ 이다. 부호 있음끼리도 같고, `iN ⊑ uM` 은 정의상 거짓이라 할 일이 없다. 핵심 보조정리는 "$a \le b$ 이면 $2^a \le 2^b$" 하나다. 증명 전체가 고등학교 지수법칙 위에 서 있다. 어려운 것은 논증이 아니라 **경우를 하나도 빠뜨리지 않는 것**이고, 그것이 기계에게 맡기는 이유다.

40.2 나눗셈과 나머지

나눗셈은 이 언어가 유별나게 조심하는 자리다.

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 mod 의 부호는 나누는 수를 따른다
fn rest input a i8 . input b i8 . output i8 . do
  return mod a b .
end
```

실행 결과

```
$ 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
```

부호 있는 나눗셈 `-7 / 2` 는 0 쪽으로 잘라 −3 이다. `-128 / -1` 은 `i8` 에 128 이 없으므로 멈춘다. C 에서는 이 한 경우가 정의되지 않은 동작이다. `mod` 의 부호는 **나누는 수**를 따른다 — `mod -7 3` 은 2, `mod 7 -3` 은 −2 다.

정리	뜻

div_unsigned_total	부호 없는 나눗셈은 0 으로 나누지 않으면 언제나 성공한다
div_signed_failure_is_only_min_neg1	부호 있는 나눗셈이 실패하는 경우는 MIN / -1 하나뿐이다
mod_sign_follows_divisor	mod 의 부호는 나누는 수를 따른다
mod_is_a_safe_index	mod h (len s) 는 언제나 0 이상 len s 미만이다

표 40.2 — 나눗셈과 나머지에 대해 증명된 것

```
부호 없는 div a b    b = 0              → 멈춘다
                     그 밖              → 언제나 값
부호 있는 div a b    b = 0              → 멈춘다
                     a = MIN 이고 b = -1  → 멈춘다 (C 에서는 미정의 동작)
                     그 밖              → 언제나 값 (0 쪽으로 자른다)
mod h n              n > 0 이면         → 0 ≤ 결과 < n  (안전한 색인)
```

마지막 정리가 실전에서 값이 크다.

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
```

실행 결과

```
$ 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%)
```

해시 테이블은 늘 mod 로 칸을 고르는데, 그 결과가 범위 안임이 증명되므로 색인 검사가 사라진다. len s 가 0 이면? 0 으로 나누므로 먼저 멈춘다. 그래서 값이 나온 경로에서는 len s > 0 이 보장된다. 논증에 빈틈이 없다.

40.3 range 가 주는 정리

매개변수의 타입 자리에 적은 range lo hi(13장)는 넓히기 이야기를 한 겹 더 정밀하게 만든다.

정리	뜻
rsub_preserves	범위끼리의 넓히기도 값을 보존한다
radd_sound	두 범위를 더한 결과는 계산된 범위 안에 있다
radd_no_check	결과 범위가 타입 안에 들어가면 넘침 검사가 필요 없다
idx_no_check	색인 범위가 길이 안이면 경계 검사가 필요 없다
rdisj_no_value	서로소인 두 범위에는 공통 값이 없다(그래서 갈래 하나가 죽는다)
derive_minimal	도출된 범위는 가장 작은 것이다(필요 이상으로 넓지 않다)

표 40.3 — 범위에 대해 증명된 것

`radd_no_check` 와 `idx_no_check` 가 검사를 지우는 최적화의 정당성이다. 그 두 정리를 실제 코드에 적용하는 분석은 41장에서 본다.

40.4 멈춤을 정직하게 모델에 넣는다

타입 규칙은 한 층 더 깊이 증명되어 있다(`LowentType.v`). 폭과 부호가 붙은 수치 식에 묶기·갈래·비교·좁히기·검사되는 산술·감싸는 산술을 얹고 세 정리를 증명했다.

- `progress` — 타입이 붙은 닫힌 식은 **값이거나, 멈추거나, 한 걸음 간다**. 막히지 않는다.
- `preservation` — 한 걸음 가도 타입이 바뀌지 않는다.
- `values_fit` — 실행이 값에 닿으면 그 값은 **언제나 그 타입의 폭 안에** 있다.

세 번째가 이 언어에서 특별하다. `u8` 자리에 8 비트를 넘는 값이 들어오면 방출된 C 가 조용히 다른 일을 한다. 그래서 범위를 타입 규칙 안에 넣고 (값에 타입을 붙이려면 범위 증거가 있어야 한다) 실행 내내 보존됨을 증명했다.

그리고 결론이 **셋**인 것이 요점이다. 검사되는 산술은 넘치면 값을 내지 않고 멈춘다. 멈춤을 값에 넣으면 정리가 거짓말이 되고, 멈춤을 빼면 정리가 거짓이 된다. 그래서 "값이거나 · 멈추거나 · 한 걸음" 이다. 멈춤을 번역 시각에 막는 것은 정상 경로의 일이고, 계약과 증명서가 그 일을 한다 (41장).

```
타입이 붙은 닫힌 식 e
   ├─▸ 값 v          values_fit: v 는 언제나 그 타입의 폭 안
   ├─▸ 멈춘다        검사되는 산술이 넘쳤다 · 0 으로 나눴다
   └─▸ 한 걸음 → e'  preservation: e' 도 같은 타입이다 (그리고 다시 셋 가운데 하나)
```

> **문. 넓히기가 실패할 수 없다는 것은 따로 증명할 만한가?**
>
> **답.** 그렇다(`widen_never_fails`). "실패할 수 없음" 이 증명되어야 넓히기 자리에 실행 중 검사를 두지 않아도 된다. 그것이 넓히기가 **공짜인 이유**다. 반대로 좁히기는 `narrow_ok_iff` 가 성공 조건을 "정확히 범위 안" 으로 못 박으므로, `narrow_try u8 300` 은 조용히 44 를 주지 않고 `none` 이다. 실패가 값이다.

> **흔한 오해. `values_fit` 가 증명되었으니 이 판의 컴파일러에서 `u8` 자리에 넘치는 값이 들어올 일은 없다**
>
> 정리는 **타입 규칙**에 대한 것이다. 컴파일러가 모든 자리에서 그 규칙을 따르는지는 따로다. 이 매뉴얼을 쓰며 찾은 예로, `pipe` 의 `map` 이 `u64` 를 내는데 `collect into` 가 `u8` 버퍼에 담는 자리를 이 판의 도구가 오래도록 조용히 감았다(지금은 `E-TYPE-COLLECT` 로 거절한다, 24장). 규칙이 옳다는 증명과 구현이 규칙을 빠짐없이 적용한다는 사실 사이의 틈이 이런 자리에서 드러나고, 그 틈은 시험과 두 백엔드 대조가 메운다. 증명은 "무엇을 막아야 하는가" 를 정하고, 막았는지는 재서 확인한다.

40.5 증명하지 않은 것

- **부동소수는 이 격자에 없다.** 정수만 다뤘다. `f32 → f64` 는 값을 보존하지만, 반대 방향과 섞인 연산은 별도 규칙이며 여기서 증명되지 않았다. 반올림·NaN·−0 도 다루지 않았다.
- **폭은 8·16·32·64 넷으로 고정했다.** 임의 폭 정수(`bits 7` 같은 것)는 구현에는 있지만(13장) 이 모델 밖이고, `u128` 은 모델에도 구현에도 없다.
- **정리는 타입 규칙에 대한 것이다.** 컴파일러가 그 규칙을 정확히 구현했는지는 시험과 두 백엔드 대조가 확인한다(31장).

복습 정리

넓히기 `⊑` 는 네 줄로 정의된 부분 순서이고, `uN ⊑ iM` 의 조건은 엄격한 `N < M` 이다. 넓히기는 값을 보존하고 실패할 수 없으며, join 은 두 피연산자 가운데 하나이고, 좁히기의 성공 조건은 정확히 범위 안이다. 부호 있는 나눗셈의 실패는 `MIN / -1` 하나뿐이고 `mod` 는 나누는 수의 부호를 따르며 언제나 안전한 색인이다. `range` 의 정리들이 검사를 지우는 근거가 되고, 타입 규칙은 "값·멈춤·한 걸음" 셋으로 멈춤을 정직하게 모델에 넣었다. 부동소수는 증명 밖이다.

41 경계의 증명 — 구간, 관계, 행우선 주소

이 장의 차례

41.1 구간으로 계산한다 466
41.2 관계 — 구간이 못 하는 것 467
41.3 네 단계 467
41.4 행우선 증명 469
41.5 증명을 신뢰하되, 신뢰를 검사로 받친다 471
41.6 증명하지 않은 것 471

먼저 알아야 할 것

9장 줄 · 계약으로 길이 조건을 적으면 색인 검사가 사라진다
14장 계약 · 강제되는 계약은 사실이 되어 검사를 지운다
39장 수학 도구상자 · 추상해석은 넓게 틀려야 하고, 반복 분석은 고정점에서 멈춘다
40장 수의 증명 · `radd_no_check` · `idx_no_check`

돌아보기

40장에서 검사를 지우는 최적화의 정당성이라고 한 두 정리는 무엇이었고, 각각 무엇을 전제로 했는가?

> **답.** `radd_no_check` 는 결과 구간이 선언 타입 안에 들면 넘침 검사가 필요 없다고, `idx_no_check` 는 색인이 0 이상이고 길이보다 작음이 알려지면 경계 검사가 필요 없다고 했다. 둘 다 **그 사실을 이미 안다** 는 것이 전제다. 이 장은 컴파일러가 그 사실을 어떻게 얻는지 — 그리고 얻었다고 잘못 믿는 일을 무엇이 막는지 — 를 다룬다.

이 장의 필요성과 맥락

이 언어는 정수 연산이 넘치면 멈추고, 색인이 범위를 벗어나도 멈춘다. 그러면 연산마다 검사가 붙어 느려질 것 같다. 실제로는 대부분의 검사가 번역 시각에 사라진다. 넘칠 수 없음, 범위 안임이 증명되기 때문이다. C 는 확인하지 않아 빠르지만 범위를 벗어나면 남의 메모리를 읽는다 — 보안 사고의 절반이 거기서 나온다. Lowent 는 셋째 길을 간다. 확인을 하되, 확인할 필요가 없음이 증명되면 그 자리에서 지운다. "정직하게 적으면 빨라진다" 는 주장이 전부 이 장의 분석에 기대므로, 그 분석이 어디서 틀릴 수 있고 무엇이 그것을 받치는지까지 본다.

이 장이 끝나면

구간 분석이 값의 범위를 계산하는 법과 그 건전성이 한 방향이라는 것을 확인한다. 구간이 못 하는 변수 사이의 사실을 한 칸짜리 관계로 나르는 법, 경계 검사가 사라지는 네 단계(직접 · 지역에 담은 길이 · 계약의 용량 · 행우선 주소)를 `--emit-proof` 출력으로 읽는다. 행우선 증명이 무너지는 조건, 분석이 틀렸을 때 VM 이 컴파일러를 고발하는 장치, 지운 검사마다 남는 증명서와 그 검산기, 그리고 계약을 붙여도 남는 검사를 보게 된다.

이 장에서 답할 질문

1 `lru_idx` 처럼 두 값이 합류한 색인은 왜 검사가 남는가?

41.1 구간으로 계산한다

컴파일러는 값 하나 대신 구간 `[lo, hi]` 로 계산해 넘침과 경계를 판정한다.

```
a ∈ [0, 100],  b ∈ [0, 100]
a + b ∈ [0, 200]        lo+lo, hi+hi
a − b ∈ [−100, 100]     lo−hi, hi−lo   뺄셈은 뒤집힌다 --- 흔한 실수 자리
a × b ∈ [0, 10000]      부호가 섞이면 네 곱의 최소·최대
```

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
```

실행 결과

```
$ 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%)
```

계약이 두 입력을 100 이하로 묶었으므로 합은 200 을 넘지 않고 `u16` 에서 넘치지 않는다. 검사가 모두 사라진다. 같은 모양이라도 입력이 묶이지 않으면 검사가 남는다.

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
```

실행 결과

```
$ 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` 은 넘치지 않지만 분석은 `a·b` 가 어디까지 커질지 모른다. 그래서 검사를 남기고, 실제 값이 들어와서야 넘치지 않음을 확인한다.

정리들은 모두 한 방향이다. **"안전하다고 말한 자리는 정말 안전하다."** 그 반대 — 안전한데 안전하다고 말하지 못함 — 는 정리가 보장하지 않는다. 그것은 성능 손해일 뿐이다.

```
분석이 넓게 잡음  →  검사가 남는다     →  느리다(안전)
분석이 좁게 잡음  →  검사가 사라진다   →  틀린 메모리 접근(위험)
```

그래서 규칙을 하나 더할 때마다 묻는 것은 늘 같다 — “이 규칙이 실제보다 좁게 잡을 수 있는가?”

41.2 관계 — 구간이 못 하는 것

구간은 변수를 따로따로 본다. `i ∈ [0, 100]`, `n ∈ [0, 100]` 을 알아도 `i < n` 인지는 모른다. 그런데 배열 안전에 필요한 사실이 바로 그것이다. 그래서 구간 옆에 **한 칸짜리 관계**를 둔다.

```
lenlt[i] = s     지역 i 는 len(지역 s) 보다 작다
lerel[i] = n     지역 i 는 지역 n 보다 작다
```

이 사실은 분기 조건에서 온다. `while lt i n` 의 몸 안에서는 `i < n` 이 참이고, 조건이 거짓인 갈래에서는 `i ≥ n` 이 참이다. 무거운 관계 영역 (팔각형·다면체) 대신 한 칸을 두는 까닭은 실제 코드의 모양 때문이다. 배열을 훑는 코드는 거의 언제나 `while lt i (len s)` 모양이고, 정교한 이론보다 실제 코드의 모양을 정확히 담는 쪽이 값이 크다.

사실을 다루는 규칙은 세 조각이다.

1. **얻기.** 분기 조건이 참인 갈래에 그 조건을 사실로 기록해 둔다.
2. **나르기.** 대입을 따라간다 — `var x be j` 에서 `j < cap` 이면 `x < cap` 이다. 합류에서는 두 경로가 같을 때만 살린다. 한쪽에서만 참인 것은 사실이 아니다.
3. **죽이기.** 관련 변수가 바뀌면 그 사실을 양방향으로 죽인다. 슬라이스가 다시 묶이면 그 길이에 대한 사실이 전부 죽는다.

셋째가 안전의 핵이다. 사실을 오래 살려 두는 것이 곧 잘못된 검사 제거이므로, 죽이는 쪽이 늘 더 공격적이다.

41.3 네 단계

경계 검사가 사라지는 모양을 세지는 순서로 늘어놓는다. `--emit-proof` 는 증명으로 지운 검사를 한 줄씩 적는다 — op 이름, 명령 위치, 연산, 쓴 규칙, 그리고 그 규칙이 쓴 범위다.

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 --- 반복 조건이 len 을 직접 본다
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 --- 길이를 지역에 담았다
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 --- 계약이 용량을 말한다
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 --- 행우선 주소 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
```

실행 결과

```
$ 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 · 직접.** `direct` 는 반복 조건이 `len s` 를 직접 본다. `index` 가 `R-IDX-LENLT` 로 지워졌다.
- **2 · 지역에 담은 길이.** `stored` 는 `let n be len s` 로 길이를 지역에 담았다. 아주 흔한 관용구다. 한때는 "이 값은 `len s` 다" 라는 사실이 스택 값의 속성이라 지역에 담는 순간 사라졌고, 이 루프의 검사가 남았다. 지역에도 그 사실을 싣도록 고친 뒤로 1 과 같이 지워진다. 이것은 안전성이 아니라 성능 수정이다 — 검사가 있었으므로 처음부터 안전했다.
- **3 · 계약의 용량.** `capacity` 는 배열 크기와 반복 상한이 다른 변수다. `j < cap`(반복 조건)이고 `len s ≥ cap`(계약)이므로 `j < len s` 다.
- **4 · 행우선 주소.** `grid` 는 `index a (add (mul i n) k)` 처럼 **계산된** 색인을 쓴다. 구간도 한 칸 관계도 곱을 다루지 못하지만 `R-IDX-ROWMAJOR` 로 지워졌다. 주소의 곱과 합도 `R-MUL-CAP`·`R-ROW-CAP` 으로 함께 증명되었다.

`i` 와 `j` 를 하나 올리는 덧셈이 `R-ADD-LENLT`·`R-ADD-LEREL` 로 함께 사라진 것도 같은 관계 덕분이다. `i < n` 이면 `i + 1 ≤ n` 이라 넘치지 않는다.

41.4 행우선 증명

> **수학. 행우선 주소가 범위 안이라는 증명**
>
> 전제는 셋이다 — 계약 `len a ≥ p·q`, 바깥 반복 조건 `i < p`, 안쪽 반복 조건 `k < q`. 주소는 멈추는 곱과 멈추는 합으로 계산한 `i·q + k` 다. 그러면 $i \le p-1$ 이므로 $i \cdot q \le (p-1) \cdot q = p \cdot q - q$ 이고, $k \le q-1$ 이므로 $i \cdot q + k \le p \cdot q - q + q - 1 = p \cdot q - 1$ 이다. 따라서 주소 $\le p \cdot q - 1 < p \cdot q \le \text{len a}$ 다. 부등식 두 개를 더했을 뿐이다.

고등학교 수학이지만, 전제가 하나라도 빠지면 무너진다.

무엇을 뺐나	색인 검사
전부 있음	사라진다
계약 `len a ≥ n·n` 을 뺀다	남는다
`i` 를 다른 변수로 묶는다(`while lt i m`)	남는다
주소를 `wrap_mul` 로 계산한다	남는다

표 41.1 — 행우선 규칙이 서는 조건

마지막 줄이 가장 미묘하다.

examples/ch41/grid_wrap.low

```
module grid_wrap .
rem run: grid [1,2,3,4] 2
rem proof

rem 같은 주소를 감기는 곱으로 계산했다
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
```

실행 결과

```
$ 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
```

같은 수식인데 `index` 줄이 없다. `wrap_mul` 은 넘칠 때 조용히 감긴다. 감긴 값은 작아질 수 있어서 $i \cdot q \le (p-1) \cdot q$ 라는 단계가 깨진다. 이 예제에서는 $n \le 1000$ 이라 실제로 감기지 않지만, 규칙은 "멈추는 곱일 때" 에만 선다. **멈추는 연산이 아니면 사실이 아니다.**

그러면 계약의 곱 $p \cdot q$ 자체가 넘치면? 계약의 곱은 멈추는 곱이고 계약 검사는 절대 지워지지 않으므로, 본문에 닿았다면 넘치지 않았다. 경계 검사를 없앤 자리의 유일한 방벽이 계약 검사다. 그래서 계약 검사는 모드로도 최적화로도 지워지지 않는다.

개발 저장소의 측정에서 이 규칙은 행렬 곱셈 벤치마크의 경계 검사를 6 개에서 1 개로 줄였고(남은 하나는 $n \ge 1$ 이 계약에 없는 `index c 0` 자리다), 계약이 용량을 말하는 LRU 벤치마크는 10 개에서 4 개로 줄였다.

흔한 오해. 검사를 줄인 만큼 빨라진다

> 검사 개수는 비용의 대리 지표가 아니다. 한 번도 걸리지 않아 분기 예측이 늘 맞는 검사는 사실상 공짜여서, 검사를 7 개에서 0 개로 줄였는데 시간이 그대로인 벤치마크가 있었다. 검사가 비싼 것은 행렬 곱셈처럼 자동 벡터화를 막는 자리다. 그래서 이 부는 개수를 줄인 것을 성과로 세지 않는다. 그리고 `--ir` 이 세는 산술 검사는 증명이 기록될 뿐 C 백엔드가 여전히 멈추는 호출을 낸다 — 그 수는 **증명된 것**이지 **사라진 것**이 아니다.

41.5 증명을 신뢰하되, 신뢰를 검사로 받친다

정리는 규칙이 옳다고 말하지만, 컴파일러가 그 규칙을 정확히 적용했는지는 따로 확인해야 한다. 장치가 둘이다.

분석 자기 고발. VM 은 검사를 지우지 않는다. 컴파일러가 "지워도 된다" 고 표시한 자리를 실행해 보고, 실제로 범위를 벗어나면 그 자리에서 `E-VM-ANALYSIS: … the interval/relational analysis is UNSOUND (this is a compiler bug)` 로 컴파일러를 고발한다. 네이티브 빌드에는 검사가 없어 빠르고, VM 에는 검사가 있어 표시가 거짓이면 알려 주며, 두 백엔드 대조가 둘을 같은 입력으로 돌린다. 이 장치가 실제로 일한 적이 있다. 행우선 규칙을 넣으면서 `requires ge (len a) (mul n n)` 을 검사되는 계약으로 믿었는데, 처리기가 그 모양의 계약을 읽지 못해 실제로는 검사가 없었고, 바로 이 진단이 떴다. **믿을 것은 검사되는 것뿐이다.**

증명서 검산. `--emit-proof` 의 줄 끝 숫자가 그 근거다. 컴파일러는 검사를 지운 자리마다 산술 근거(Farkas 증명서)를 남기고, 컴파일러와 따로 짠 검산기가 그 산술을 다시 계산한다. 첫 줄의 `certified` 수가 검산을 통과한 수다. 검산기 가운데 하나는 규칙의 건전성을 증명한 Coq 코드에서 OCaml 로 **추출한** 것이다(`LowentCert.v`·`LowentCertExtract.v`). 사람이 옮긴 검산기는 "이 산술을 옳게 옮겼나" 를 아무도 확인하지 않지만, 추출본은 정리가 말하는 함수 그 자체다. 다만 이것은 규칙의 **산술**을 믿을 만하게 할 뿐, "그 자리에 그 사실이 정말 성립하는가" 는 검산기의 관할 밖이다.

> **실제 사례. 멈추지 않는 도구**
>
> 구간 분석의 예제를 만들다가, 입력 `n` 만큼 도는 반복이 든 op 에서 `--check` 는 곧 끝나는데 `--ir` 이 60 초를 넘겨도 끝나지 않는 일이 있었다. `--ir` 은 계약에서 경계값을 뽑아 op 을 실제로 실행해 보는데, `n = 2^64 - 1` 이면 그 반복은 사람의 수명 안에 끝나지 않는다. 그 실행에 걸음 예산을 두고, 예산에 걸린 경우는 **건너뛰었다고 센다**. 조용히 통과시키지 않는 것이 요점이다. 검사하지 않은 것을 검사했다고 말하는 순간 그 도구는 거짓말이 된다. 사용자가 돌리는 `--run` 에는 예산이 없다 — 끊어도 되는 것은 도구가 스스로 만든 시험뿐이다.

> **문. `lru_idx` 처럼 두 값이 합류한 색인은 왜 검사가 남는가?**
>
> **답.** 합류에서는 두 경로가 같을 때만 사실이 산다. `lru_idx` 가 한쪽에서는 `0`, 다른 쪽에서는 `j` 로 들어오면, `j < cap` 이라는 사실은 한쪽 경로의 것 이라 합류에서 죽는다. `0 < cap` 을 따로 알더라도 한 칸 관계는 "둘 다 cap 보다 작다" 를 모아 나르지 않는다. 사실을 너무 오래 살려 두는 쪽으로 틀리는 것보다는 검사를 남기는 쪽으로 틀리는 것이 이 분석의 방향이다.

41.6 증명하지 않은 것

계약을 붙여도 빠지지 않는 검사가 남는다. 측정한 자리들이다.

남은 자리	왜 못 지우나
행렬 곱셈의 `index c 0`	`n ≥ 1` 이 계약에 없다(더하면 닫힌다)
LRU 의 `index keys lru_idx`	`0` 과 `j` 가 합류한 값이라 관계 사실이 합류에서 죽는다
체의 `index s i`(`while lt (mul i i) n`)	`i·i < n ⇒ i < n` 은 또 다른 비선형 모양이고, 실제 코드에 드물어 규칙을 만들지 않았다
정렬의 파티션 색인	`hi ≤ len s` 를 비엄격으로 나르는 자리가 없다(지금 관계는 엄격 전용이다)

표 41.2 — 남는 검사와 그 까닭

- **위드닝은 정보를 버린다.** 고정점에 빨리 닿으려고 넓게 잡으므로 안전한 자리에서도 검사가 남는다(39장의 `count_up`). 손해이지 위험이 아니다.
- **배열 내용에 대한 사실은 이 영역에 자리가 없다.** "모든 원소가 10 보다 작다" 는 어떤 `requires` 로도 닫히지 않는다. 도구의 한계이지 프로그램의 결함이 아니고, 검사가 남으므로 답은 여전히 옳다.
- **부동소수 구간은 다루지 않는다.**
- **분석 규칙과 정리 사이의 대응은 경험적이다.** 자기 고발·증명서 검산·두 백엔드 대조가 받친다(50장).

복습 정리

구간 분석은 값 대신 범위로 계산하고, 넓게 틀리면 느릴 뿐 좁게 틀리면 위험하다. 한 칸짜리 관계가 분기 조건에서 `i < len s` 같은 사실을 얻어 나르고, 관련 변수가 바뀌면 공격적으로 죽인다. 경계 검사는 직접 · 지역에 담은 길이 · 계약의 용량 · 행우선 주소의 네 단계에서 사라지고, `--emit-proof` 가 그 규칙을 한 줄씩 적는다. 행우선 증명은 멈추는 곱일 때만 서고, 계약 검사는 절대 지워지지 않는다. VM 의 자기 고발과 Coq 에서 추출한 증명서 검산기가 신뢰를 받치며, 합류·비선형·배열 내용 때문에 남는 검사는 남는다고 적는다.

42 소유와 차용의 증명

이 장의 차례

42.1 두 기계와 세 사건 474
42.2 중심 정리 474
42.3 거절되는 모양, 통과하는 모양 475
42.4 모델을 넓히다가 구현의 결함을 찾았다 475
42.5 모델과 구현 사이 476
42.6 증명하지 않은 것 477

먼저 알아야 할 것

12장 빌리기 · 읽기 여럿 또는 쓰기 하나
19장 소유 · 메모리 규칙의 강도 — 정적·동적·증명됨
39장 수학 도구상자 · 불변식과 귀납법 — 한 걸음이 참을 참으로 넘긴다

돌아보기

12장의 `stale.low` 는 `r` 로 `n` 을 읽기로 빌린 채 `n` 에 5 를 썼다. 무엇으로 거절되었고, 그 규칙은 무엇을 지키는가?

> **답.** `E-EXCL` 로 거절되었다. 한 값에 대한 빌림은 읽기 여럿 또는 쓰기 하나여야 하고, 그 규칙은 **읽는 쪽의 믿음** — 자기가 보는 동안 값이 바뀌지 않는다는 것 — 을 지킨다. 이 장은 컴파일러가 이 규칙으로 "통과" 시킨 프로그램이 정말로 실행 중 위반이 없는지를 다룬다.

이 장의 필요성과 맥락

차용 규칙은 이 언어의 메모리 안전과 데이터 경합 부재의 바탕이다. 그런데 규칙이 있다는 것과 규칙이 **충분하다**는 것은 다르다. 규칙에 구멍이 있으면 컴파일러의 "통과" 는 아무 뜻이 없다. 이 장의 정리는 그 "통과" 에 뜻을 준다. 그리고 증명 과정에서 실제 컴파일러의 결함 하나가 정리로 옮겨졌다. 증명이 무엇을 하는지 가장 잘 보여 주는 이야기라서, 수와 경계 다음 자리에 둔다.

이 장이 끝나면

정적 기계(살아 있는 빌림 목록)와 동적 기계(빌림 스택)라는 두 모델, 그리고 프로그램을 세 가지 사건의 나열로 줄이는 방법을 알게 된다. 중심 정리 "정적 검사를 통과한 프로그램은 실행 중 차용 위반이 없다" 와 그 대우, 증명에 필요했던 두 불변식을 이해한다. 칸(place)과 op 경계로 모델을 넓힌 이야기, 거절되는 모양과 통과하는 모양, 모델과 구현 사이를 받치는 천만 개 규모의 전수 검사도 보게 된다. 반복은 43장에서 다룬다.

이 장에서 답할 질문

1 칸 모델이 증명되었다면 `mut_ref (field s a)` 와 `mut_ref (field s b)` 를 동시에 쓸 수 있는가?

42.1 두 기계와 세 사건

같은 규칙을 보는 기계가 둘 있다. **정적** 기계는 컴파일할 때 살아 있는 빌림의 목록을 들고, 겹치는 빌림을 만들려 하면 거절한다. **동적** 기계는 실행 중에 자리마다 빌림 **스택**을 두고, 무효가 된 빌림을 쓰면 멈춘다. 이 장의 정리는 두 기계가 서로 어긋나지 않는다는 것이다.

증명을 위해 프로그램을 세 종류 사건의 나열로 줄인다. 전체 언어 대신 본질만 남긴 작은 대상을 만드는 것이 증명의 첫 기술이다.

사건	뜻
`Create t x mut`	자리 `x` 에 대한 빌림을 태그 `t` 로 만든다. `mut` 는 쓸 수 있는지
`Use t wr`	빌림 `t` 로 접근한다. `wr` 는 쓰기인지
`Own x wr`	소유자가 자리 `x` 에 직접 접근한다

표 42.1 — 사건 셋

동적 기계는 규칙 다섯으로 정의된다. 빌림을 만들면 스택에 쌓고(D1), 빌림으로 **읽으면** 그 위의 쓰기 빌림만 버리고 읽기 빌림은 남기며(D2), 빌림으로 **쓰면** 그 위를 전부 버린다(D3). 소유자가 읽으면 쓰기 빌림을 모두 무효로 하고(D4), 소유자가 쓰면 스택을 비운다(D5). D2 와 D3 의 차이가 요점이다 — 읽기는 여러 독자가 공존해도 문제가 없지만, 쓰기 뒤에는 위의 이름들이 낡은 값을 보므로 더는 쓸 수 없어야 한다. 그리고 소유자가 다시 손대는 것 자체를 오류로 만들지 않고, 그 뒤에 죽은 빌림을 **쓰려 할 때** 잡는다.

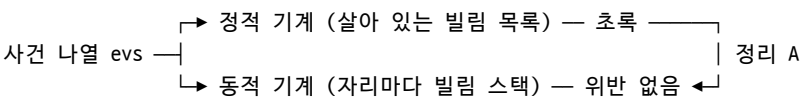
사건 넷을 차례로 먹이면 자리 `x` 의 스택이 이렇게 바뀐다.

```
사건                      스택 (아래 → 위)
Create r1 x 읽기   D1     r1
Create w2 x 쓰기   D1     r1 w2
Use r1 읽기        D2     r1          r1 위의 쓰기 빌림 w2 를 버린다
Use w2 쓰기               → 위반       w2 는 이미 스택에 없다
```

정적 기계는 둘째 사건에서 이미 거절한다. 읽기 빌림 `r1` 이 살아 있는데 같은 자리에 쓰기 빌림을 만들기 때문이다.

42.2 중심 정리

> **수학. 정리 A — 합의(`LowentEXCL.v` 의 `agreement`)**
>
> `forall evs, wf evs -> static_green evs = true -> dyn_clean evs = true.` 잘 짜인 모든 사건 나열에 대해, 정적 검사를 통과하면 동적 기계에서도 위반이 없다. 대우로 읽으면(`no_violation_in_green`) 실행 중에 차용 위반이 났다면 정적 검사는 반드시 그 프로그램을 거절했다.




```
정적 초록 ⇒ 동적 위반 없음      (증명됨)
동적 위반 없음 ⇒ 정적 초록      (참이 아니다 – 정적 검사는 보수적이다)
```

대우가 실용적인 뜻을 선명하게 한다. **번역을 통과한 코드에서 실행 중에 차용 위반을 보는 일은 없다.** "번역은 됐는데 돌려 보니 빌림이 깨졌다" 는 조합 자체가 이 모델 안에서는 일어날 수 없다.

반대 방향 — 동적으로 안전하면 정적으로도 통과한다 — 은 참이 아니고 참일 필요도 없다. 정적 검사는 보수적이라 실제로는 안전한 프로그램도 거절할 수 있다. 안전 쪽으로 틀리는 것은 불편이고, 위험 쪽으로 틀리는 것은 재앙이다. 41장의 구간 분석과 같은 비대칭이다.

증명은 불변식 보존이고, 불변식이 둘 필요했다.

- **INV** — 정적으로 살아 있는 빌림은 동적 스택에도 있고, 대상 자리와 쓰기 여부가 일치한다.
- **SINV** — 같은 자리에 살아 있는 서로 다른 빌림 둘은 모두 읽기 빌림이다.

SINV 가 왜 필요했는가. 빌림으로 쓰면(D3) 스택의 위를 전부 버리는데, 버려진 것 가운데 정적으로는 아직 살아 있는 빌림이 있으면 INV 가 깨진다. "쓰기 빌림은 하나뿐" 이라는 규칙이 겹침을 막으므로, 쓰기를 하는 빌림과 같은 자리에 살아 있는 다른 빌림은 없다. 직관은 "겹치면 위험하니까" 라고 말하지만, 증명은 더 정확한 이유를 준다 — **그 규칙이 없으면 두 기계의 대응이 무너진다.** 규칙을 느슨하게 하면 정확히 어디가 깨지는지 증명이 가리킨다.

나머지는 자료구조에 대한 지루한 보조정리다 — "담은 자리에서 꺼내면 담은 것이 나온다"(`get_set_same`), "다른 자리는 바뀌지 않는다" (`get_set_other`), "스택의 위를 버려도 그 태그는 남는다"(`in_pop_above`). 증명이 길어지는 곳은 늘 흥미로운 곳이 아니라 이런 곳이다.

42.3 거절되는 모양, 통과하는 모양

정리가 막는 것을 코드의 모양으로 옮기면 이렇다. 네 모양 모두 12장의 예제로 실제 진단을 보았다.

모양	판정	사건 모델에서
쓰기 빌림이 살아 있는데 소유자가 쓴다	거절(`E-EXCL`)	D5 가 스택을 비워, 그 뒤의 빌림 사용이 위반이다
같은 값을 쓰기로 두 번 빌린다	거절(`E-EXCL`)	겹친 빌림 가운데 쓰기가 있으면 SINV 가 깨진다
읽기 빌림으로 쓴다	거절(`E-TYPE-REF`)	읽기 빌림은 쓰기 사건을 낼 수 없다
읽기 빌림 여럿으로 함께 읽는다	통과	D2 는 읽기 빌림을 죽이지 않는다 — 위험이 없다

표 42.2 — 빌림의 모양과 정리

42.4 모델을 넓히다가 구현의 결함을 찾았다

평평한 모델은 자리를 번호로 봤다. 실제 언어는 더 복잡하다. 모델을 두 번 넓혔고, 두 번 다 정리 A 를 다시 증명했다.

칸(place). 평평한 모델에서 `x.a` 와 `x.b` 는 둘 다 그냥 `x` 라서, 서로 다른 칸의 쓰기 빌림 둘이 충돌하는 것으로 나온다. 실제 규칙은 통과시킨다. 증명이 구현을 따라가지 못하면 그 증명은 다른 언어에 대한 것이다. 그래서 자리를 경로로 일반화하고, 자리의 **같음**을 **겹침**("한쪽이 다른 쪽의 접두사다")으로 바꿨다. 정리와 불변식의 구조는 그대로였다. 좋은 추상을 골랐으면 확장이 이렇게 국소적으로 끝난다.

op 경계. 사건 언어에는 호출이 없었다. 그래서 op 이 인자로 받은 빌림을 그대로 참조로 돌려주어 빌림을 **세탁**하는 모양이 모델 밖에 있었다. 구현이 그것을 실제로 겪었다 — 정적 검사가 돌려받은 참조를 원래 빌림과 같은 것으로 알아보지 못해 통과시켰다. 결함을 고친 뒤 그 이야기를 정리로 옮겼다.

> **수학. 옛 규칙은 건전하지 않았다(`LowentLaunder.v`)**
>
> `old_rule_is_unsound` — 세탁 사례에서 옛 정적 검사는 통과시키는데 동적 기계는 위반을 낸다. `new_rule_catches_it` — 고친 규칙은 그 사례를 거절한다. 그리고 `honest_laundering` — 정직한 반환은 여전히 통과한다. 마지막 정리가 없으면 "모두 거절" 도 정답이 된다.

그때까지 그것은 "우리가 겪었다" 는 **이야기**였다. 이야기는 잊히고 리팩터링에 지워진다. 이제 그것은 **정리**다. 누가 그 검사를 지우면 증명이 깨진다. 이것이 이 저장소에서 증명이 하는 가장 값진 일이다.

> **문. 칸 모델이 증명되었다면 `mut_ref (field s a)` 와 `mut_ref (field s b)` 를 동시에 쓸 수 있는가?**
>
> **답.** 정적 검사는 통과시킨다. 그러나 이 판의 중간 표현은 지역이 아닌 자리(칸)를 가리키는 참조를 아직 내리지 못해서, 실행하려 하면 `E-IR-UNSUP·E-VM-UNSUP` 으로 못 한다고 말한다. 증명된 규칙과 구현된 기능은 다른 것이다. 게다가 이 판의 `--check` 는 그 진단을 찍으면서도 "check: ok" 로 끝나는데, 오류 코드가 있는데 초록인 것은 결함이다.

42.5 모델과 구현 사이

정리는 종이 위의 두 기계에 대한 것이다. 그 기계가 실제 컴파일러와 같은지는 **유계 전수 모델 검사**가 받친다. 자리 둘·태그 넷 이하·길이 7 이하의 모든 사건 나열 10,490,024 개를 구현의 정적 검사와 동적 기계에 먹여 판정이 모델과 같은지 본다. 위반은 0 이다. 옵트인으로 길이 9 의 13 억 개도 돌렸고 위반 0 이다.

길이	사건 나열	시간	위반
6	920,918	4.3 초	0
7 — 기본	10,490,024	5.0 초	0
8 — 옵트인	119,892,870	13.4 초	0
9 — 옵트인	1,366,571,808	99 초	0

표 42.3 — 전수 검사의 봉투(단위 시험 전체 시간으로 잼)

봉투를 열한 배 넓히는 데 든 비용이 1 초가 되지 않았다. "비싸서 못 한다" 고 적기 전에 재 봐야 한다. 그래도 길이 10 이상은 **모른다** — 그리고 그것을 적는 것이 이 절의 일이다.

반복의 분기 합류는 정리 밖이고, 반복 체커가 90,376 형태 × 반복 1 … 3 을 전수 확인한다(43장). 이것들은 "증명됨" 이 아니라 "전수 검사됨" 이다.

> **흔한 오해. 전수 검사에서 위반이 0 이면 증명된 것이다**
>
> 전수 검사는 **봉투 안**에서만 말한다. 길이 9 까지 13 억 개를 돌려 위반 0 이어도 길이 10 은 모른다. 그래서 이 장은 정리 A·B 에는 "증명됨" 을, 모델과 구현의 대응에는 "전수 검사됨" 을 따로 적는다. 둘을 섞어 부르면 봉투 밖의 결함이 "증명된 영역" 에 숨는다. 이 매뉴얼을 쓰며 찾은 `let` 이름을 `mut_ref` 로 빌려주면 값이 바뀌는 결함(12장)도, 불변 이름이라는 개념 자체가 이 사건 모델에 없어서 정리와 전수 검사 둘 다 닿지 않는 자리다.

42.6 증명하지 않은 것

- **모델은 직선 사건 나열이다.** 갈래와 반복이 없다. 실제 컴파일러는 제어 흐름을 따라 빌림의 생사를 계산하는데, 정리 A 는 그 계산이 옳다는 것을 말하지 않는다. 그것이 43장의 주제다.
- **태그가 신선하다고 가정한다**. 구현은 태그를 재사용하지 않지만 그 사실 자체는 증명 밖이다.
- **메모리 자체를 모델링하지 않았다**. 사건 모델은 “누가 접근할 권리가 있나” 만 본다. 그 권리 체계가 해제 후 사용 같은 실제 메모리 안전으로 이어지는 단계는 별도다.
- **순차 단편**에 대한 정리다. 여러 흐름으로의 확장은 45장의 데이터 경합 정리가 맡는다.

> **복습 정리**
>
> 정적 기계(빌림 목록)와 동적 기계(빌림 스택)를 세 사건의 나열 위에서 맞대어, 정적 검사를 통과한 프로그램은 실행 중 차용 위반이 없다는 정리 A 를 증명했다. 대우로 읽으면 번역을 통과한 코드에서 실행 중 차용 위반은 없다. 증명에는 두 기계의 대응(INV)과 “겹친 빌림은 모두 읽기”(SINV)가 필요했다. 칸과 op 경계로 모델을 넓히며 구현의 세탁 결함을 정리로 옮겼다. 모델과 구현 사이는 천만 개 규모의 전수 검사가 받친다.

43 되풀이와 고정점 — “몇 번을 돌든” 을 증명하기

이 장의 차례

43.1 두 번째 회차에서 깨진다 478
43.2 고정점이면 몇 번이든 480
43.3 위험한 모양은 하나뿐이다 480
43.4 신선한 토큰 — 이름 문제를 없애는 한 수 480
43.5 중첩 반복 — 함수를 관계로 481
43.6 증명하지 않은 것 482

먼저 알아야 할 것

7장 흐름 · `while` 은 조건이 참인 동안 몸을 되풀이한다
39장 수학 도구상자 · 고정점은 한 번 더 적용해도 변하지 않는 점이다
42장 소유와 차용의 증명 · 정리 A 는 직선 사건 나열에 대한 것이다

돌아보기

42장의 정리 A 에서 “증명하지 않은 것” 의 첫 줄은 무엇이었는가?

> **답.** 모델이 직선 사건 나열이라 갈래와 반복이 없다는 것이었다. 실제 컴파일러는 코드를 **한 번** 보는데 그 코드는 **백만 번** 돌 수 있다. 한 번의 검사로 백만 번을 보장해야 한다. 이 장이 그 방법이고, 답은 한 낱말이다 — 고정점.

이 장의 필요성과 맥락

반복이 위험한 까닭은 분명하다. 짧게 돌면 괜찮은데 오래 돌리면 깨지는 결함은 가장 찾기 어렵다. 시험은 짧게 돌리기 때문이다. 그리고 반복 안에서는 같은 이름이 회차마다 다른 것을 가리킨다 — 1 회차의 빌림과 2 회차의 빌림은 소스에서 같은 `r` 이지만 실행에서는 다른 빌림이다. 이 장은 그 두 어려움을 고정점과 신선한 토큰으로 풀고, 반복에서 빌림이 깨지는 모양이 **정확히 하나**임을 보인다. 규칙을 몇 개나 만들어야 하는지를 감이 아니라 논증으로 아는 사례다.

이 장이 끝나면

반복을 넘어 사는 빌림이 두 번째 회차에서 깨지는 모양과, 회차 안에서 태어나 끝나는 빌림이 안전한 모양을 예제로 확인한다. 정적 상태가 한 바퀴 뒤 되돌아오면 몇 번을 돌든 안전하다는 정리 B, 회차마다 새 토큰을 내어 이름 문제를 없애는 방법, 증명에 더해진 불변식 FRESH 를 알게 된다. 중첩 반복을 관계로 적어 회차마다 달리 도는 반복까지 덮은 일반화와, 종료를 다루지 않는 까닭도 보게 된다.

이 장에서 답할 질문

1 컴파일러는 고정점을 어떻게 찾는가? 정리가 그것까지 보장하는가?

43.1 두 번째 회차에서 깨진다

```
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
```

실행 결과

```
$ 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)
```

반복 밖에서 만든 읽기 빌림 `r` 을 반복 안에서 쓰면서, 같은 반복 안에서 소유자가 `x` 에 쓴다. 1 회차는 문제가 없어 보인다. 그러나 그 쓰기가 `r` 을 죽이고, 2 회차의 `deref r` 이 죽은 빌림을 쓴다. 한 번 돌 때는 멀쩡하고 두 번째에 터지는 모양이다. 처리기는 이것을 번역에서 거절한다.

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
```

실행 결과

```
$ lowentc --run drift loop_fresh.low 5
drift(5) = 6
```

반대로 빌림이 반복 **안에서 태어나 안에서 끝나면** 회차마다 신선하고, 회차 사이에 낡은 것이 남지 않는다. “반복 안에서 만들고 반복 안에서 끝내라” 는 흔한 조언에 증명이 붙은 것이다.

43.2 고정점이면 몇 번이든

여기서 함수 `f` 는 “반복 몸을 한 번 도는 것” 이고, 값 `x` 는 **정적 검사기의 상태**(살아 있는 빌림 목록)다. 몸을 한 번 돌아도 정적 상태가 그대로면 그것이 고정점이다. 그러면 귀납법이 선다.

```
0 번 돌면 안전하다.                                  기초 --- 아무 일도 안 했다
상태 L 에서 안전하게 한 번 돌면 상태가 다시 L 이다.      보존
──────────────────────────────────────
따라서 몇 번을 돌아도 안전하다.                       ∀k
```

도미노를 세울 때 조각의 간격이 모두 같아야 하는 것과 같다. 상태가 회차마다 달라지면 “다음 조각” 을 논증할 수 없다. 상태가 되돌아오니까 같은 논증을 끝없이 다시 쓸 수 있다.

수학. 정리 B — 반복 합의(`LowentLoop.v` 의 `loop_agreement`)

> `forall pre body, (exists lpre, srun [] pre = Some lpre /\ srun lpre body = Some lpre) -> forall k, dyn_clean pre body k = true`. 준비 코드 `pre` 를 지난 정적 상태에서 반복 몸 `body` 를 한 번 돌아 **같은 상태로 돌아오면**, 그 반복은 몇 번을 돌든 차용 위반이 없다. `forall k` 가 이 정리의 전부다 — 0 번, 1 번, 2^{64} 번 모두 다. `k = 0` 도 들어간다. 사소해 보이지만 빼먹으면 귀납이 서지 않는다.

고정점에 닿지 않는 코드도 있다. 회차마다 빌림이 하나씩 늘고 죽지 않아 반복 밖으로 새면 정적 상태가 자란다. 그런 코드는 정리의 전제를 만족하지 않고, 처리기가 거절한다. 빌림이 몸 안에서 태어나 몸 안에서 죽으면 상태가 되돌아온다 — 그것이 보통의 코드다.

43.3 위험한 모양은 하나뿐이다

증명을 하며 알아낸 사실이 있다. 반복에서 빌림이 깨지는 방식은 **한 가지뿐**이다.

```
pre 에서 만든 빌림 τ 를 body 에서 쓰고, 동시에 body 안에 소유자 접근 own(x) 가 있다.
  → i 회차의 own(x) 가 τ 를 죽인다.
  → i+1 회차의 use(τ) 가 위반을 낸다.
```

정적 규칙 하나가 정확히 그 짝 — 반복을 넘어 사는 빌림 × 몸의 소유자 접근 — 을 거절하고, 정리 B 의 증명은 “그 하나로 충분하다” 를 보인다. 다른 위험은 없다. 이것이 증명의 **실용적 산출물**이다. 규칙이 모자라면 결함이 새고, 넘치면 정상 코드가 막히는데, 몇 개가 맞는지를 논증이 정했다.

43.4 신선한 토큰 — 이름 문제를 없애는 한 수

반복의 진짜 골치는 **이름 충돌**이다. 1 회차의 `r` 과 2 회차의 `r` 은 소스에서 같은 이름인데 실행에서는 다른 빌림이다. 보통 이것을 다루려면 “태그 재명명” 이라는 장치를 만든다.

이 증명은 그 장치를 만들지 않았다. 대신 모델이 **실행할 때마다 새 토큰을 발행한다**. 정적 태그 `t` 는 환경을 거쳐 “지금의 토큰” 을 가리키고, 그 토큰은 회차마다 새 번호다. 그러면 지난 회차의 토큰은 아무도 가리키지 않으므로 저절로 무효가 된다. 이것은 억지가 아니다 — **구현이 실제로 신선한 태그를 발행한다.** 모델이 구현을 따라간 덕에 증명이 짧아졌다. 증명이 어려우면 모델을 다시 보라. 어려움의 절반은 잘못된 표현에서 온다.

증명에는 42장의 INV·SINV 에 불변식이 하나 더 붙는다.

- **FRESH** — 발행된 토큰은 모두 `next` 보다 작다.

이것이 없으면 새 토큰이 옛 토큰과 같은 번호를 받을 수 있고, 그러면 죽은 빌림이 **되살아난다**. 증명에서 가장 무서운 종류의 구멍이다 — 안전 쪽이 아니라 위험 쪽으로 틀린다.

43.5 중첩 반복 — 함수를 관계로

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 안쪽 반복은 바깥 회차마다 다른 횟수로 돈다(0, 1, 2, 3 번)
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
```

실행 결과

```
$ lowentc --run tri nested.low 4
tri(4) = 6
```

안쪽 반복은 바깥 회차마다 0, 1, 2, 3 번 돈다. 빌림 `mut_ref x` 는 안쪽 몸 안에서 태어나 끝나므로 어느 회차에서도 안전하다.

평평한 정리 B 는 `unroll body k` — "모든 회차가 똑같이 돈다" — 를 강요한다. 실제 프로그램은 그렇지 않다. 중첩 반복의 증명(`LowentNest.v`)은 프로그램을 나무(`NLeaf·NSeq·NLoop`)로 두고, 펼침을 **함수가 아니라 관계**로 적었다.

> **수학. 중첩 반복(`nested_agreement`)**
>
> 정적 검사를 통과한 중첩 반복 프로그램은 **어떤 펼침에서도** 차용 위반이 없다. 깊이 제한이 없고, 반복 횟수 제한이 없고, **회차마다 달리 돌아도 된다**. 관계로 적으면 "1 회차의 안쪽은 세 번, 2 회차는 0 번" 이 그냥 같은 나무의 다른 펼침이다. 그리고 옛 평평한 정리는 따름정리로 되찾는다 (`flat_loop_recovered`). 되찾지 못하면 그것은 일반화가 아니라 다른 정리다.

첫 회차가 상태를 바꾸는 반복도 들어온다. `b ; loop b` 로 한 번 벗기면 그 뒤로는 고정점이 된다(`peeled_loop_is_safe`).

> **문. 컴파일러는 고정점을 어떻게 찾는가? 정리가 그것까지 보장하는가?**
>
> **답.** 정리는 들어가는 자리에서 고정점을 **전제**로 요구한다. 컴파일러는 갈래와 회차의 상태를 합류시켜(격자의 join 과 단조 반복으로) 그 자리에 **닿는데**, 그 닿는 과정은 모델 밖이다. 그 간극은 유계 전수 검사가 받친다 — 반복 체커가 90,376 형태를 회차 1 … 3 으로 모두 돌려 모델과 구현의 판정이 같은지 본다. 이것은 "증명됨" 이 아니라 "전수 검사됨" 이다.

> **흔한 오해. 반복 몸을 한 번 따라가 보아 문제가 없으면 몇 번을 돌아도 괜찮다**
>
> `loop_stale.low` 가 반례다. 1 회차는 멀쩡하고 2 회차에서 죽은 빌림을 쓴다. "한 번 따라가 보기" 가 충분한 것은 몸을 한 번 돈 뒤의 정적 상태가 **처음과 같을 때**, 곧 고정점일 때뿐이다. 그래서 검사기는 한 번이 아니라 상태가 더 바뀌지 않을 때까지 몸을 되풀이해 적용하고, 이 장의 정리가 그 멈춤이 옳다는 것을 보장한다. 사람이 코드를 읽을 때도 같다 — "둘째 바퀴가 시작할 때 무엇이 살아 있는가" 를 묻는다.

43.6 증명하지 않은 것

- **갈래(`if`)의 합류는 정리 B 안에 없다.** 모델은 (`pre`, `body`) 두 직선 나열이다. 합류의 정확성은 위의 전수 검사가 받친다.
- **고정점에 닿는 과정**은 모델 밖이다. 정리는 닿은 뒤를 말한다.
- **반복이 끝나는지는 다루지 않는다.** 정리는 "돌면 안전하다" 이지 "끝난다" 가 아니다. 무한 반복은 펌웨어의 주 반복처럼 이 언어에서 정당한 프로그램이다. 그래서 도구가 스스로 만든 시험을 돌릴 때는 걸음 예산을 둔다(41장) — 종료를 보장할 수 없으니 도구가 멈출 줄 알아야 한다.

> **복습 정리**
>
> 반복을 넘어 사는 빌림과 몸의 소유자 접근이 만나면 두 번째 회차에서 깨지고, 처리기는 그 짝 하나를 거절한다. 몸 안에서 태어나 끝나는 빌림은 회차마다 신선하다. 정적 상태가 한 바퀴 뒤 되돌아오면(고정점) 몇 번을 돌든 차용 위반이 없다는 정리 B 는 `k` 에 대한 귀납이고, 회차마다 새 토큰을 내는 모델과 불변식 FRESH 가 이름 문제를 없앤다. 중첩 반복은 펼침을 관계로 적어 회차마다 달리 도는 반복까지 덮었다. 합류와 고정점에 닿는 과정은 전수 검사가 받치고, 종료는 다루지 않는다.

44 효과의 증명 — “무엇을 할 수 있나” 를 타입에 적기

이 장의 차례

44.1 효과는 원자의 집합이다 484
44.2 처리기가 강제하는 네 규칙 484
44.3 권한 — 효과의 짝 486
44.4 효과 건전성 — 순환이 아니다 486
44.5 효과에 기댄 최적화는 적법하다 487
44.6 증명하지 않은 것 487

먼저 알아야 할 것

15장 효과 · 효과는 호출을 따라 번지고, 순수함은 차례 바꾸기·기억하기·지우기를 허락한다
16장 권한 · 효과는 무슨 일인지, 권한은 누가 허락했는지를 적는다
39장 수학 도구상자 · 격자와 join

돌아보기

15장에서 `fn` 과 `proc` 을 가르는 기준은 무엇이었고, 순수하다는 약속은 처리기에게 무엇을 허락했는가?

> **답.** `fn` 은 바깥에 자국을 남기지 않고 `proc` 은 적은 효과만큼 남긴다. 순수한 op 은 같은 인자에 같은 답을 내므로 처리기가 호출의 차례를 바꾸거나, 결과를 기억하거나, 쓰이지 않는 호출을 지워도 된다. 이 장은 그 약속이 **정말로 지켜진다**는 증명 — 그리고 그 증명이 왜 순환 논증이 아닌지 — 를 다룬다.

이 장의 필요성과 맥락

40·43장은 “메모리를 어떻게 만지나” 였다. 이 장은 “무엇을 할 수 있나” 다. `effects none` 은 강한 약속이다 — 전역을 읽지도, 할당하지도, 화면에 쓰지도, 기다리지도 않는다. 그 약속을 최적화기가 믿고 호출을 지우는데 실제로는 파일을 썼다면, 관측할 수 있는 동작이 사라진다. 효과 체계는 메모리 안전을 주지 않는다. 대신 다른 종류의 안전 — 최적화의 정당성, 감사할 수 있음, 작은 기계에서의 거절 — 을 준다. 그 안전이 선언을 믿는 데서 오므로, 선언이 실제를 덮는다는 정리가 필요하다.

이 장이 끝나면

효과가 원자의 집합이고 부분집합 순서로 격자를 이룬다는 것, 전파와 게이팅이 한 격자의 두 연산이라는 것을 알게 된다. 처리기가 강제하는 네 규칙 (전파 · 순수성 · 원자의 집합 · 멈춤은 효과가 아니다)을 예제로 확인하고, 권한이 효과의 짝으로서 주는 감쇠와 국소 감사를 익힌다. 효과 건전성 정리가 순환이 아닌 까닭, 효과에 기댄 네 최적화가 적법하다는 증명과 그 반례도 보게 된다.

이 장에서 답할 질문

1 예약된 낱말을 매개변수 이름으로 쓰면 왜 효과 진단이 아니라 이름 진단이 나오는가?

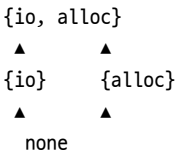
44.1 효과는 원자의 집합이다

효과는 **원자의 집합**이다. `effects io alloc .` 처럼 여러 원자를 가질 수 있다. 그러면 자연스러운 순서가 하나 있다 — $\varepsilon_1 \sqsubseteq \varepsilon_2$ 는 $\varepsilon_1 \subseteq \varepsilon_2$, 곧 "$\varepsilon_1$ 이 ε_2 보다 적게 한다" 이다. 부분집합 관계는 격자를 이루고, join 은 합집합이며, `none`(빈 집합)이 가장 아래다.

무엇	격자 연산	뜻
전파	⊔(합집합)	내가 부르는 것들의 효과를 모두 모으면 내 효과다
게이팅	⊆(부분집합)	선언한 것보다 적게 하는 것은 괜찮다

표 44.1 — 격자 하나가 하는 두 가지 일

40장의 타입 격자와 **같은 모양**이다. 다른 문제에 같은 수학이 답한다. 그리고 작은 기계의 판정이 ⊆ 한 번이 된다. 힙이 없는 기계에서 `alloc` 하는 코드는 번역되면 안 되는데(30장), 효과 선언이 있으면 "이 op 의 효과가 이 등급이 감당하는 집합에 들어가는가" 를 물으면 끝이다.




```
전파    f 가 g(effects io)와 h(effects alloc)를 부르면
       effects(f) ⊇ {io} ⊔ {alloc} = {io, alloc}      모자라면 E-EFFECT
게이팅  힙 없는 보드가 감당하는 집합에 alloc 이 없으면
       {io, alloc} ⊄ 그 집합                          그 보드로는 번역되지 않는다
```

`io` 를 `{read, write}` 로 쪼개는 **계층** 격자도 쓸 수 있다. 이 언어는 지금 **평면**을 쓴다. 계층으로 얻는 정밀도는 권한 쪽에서 이미 얻을 수 있고, 평면은 ⊆ 판정이 간단해 비용이 보인다. 정밀도를 두 곳에서 표현하면 그 둘이 어긋나기 시작한다.

44.2 처리기가 강제하는 네 규칙

규칙 1 — 전파. `f` 가 `g` 를 부르면 `effects(f) ⊇ effects(g)` 여야 한다. 아니면 `E-EFFECT`(`fn` 이면 `E-EFFECT-CALC`)다. `effects none` 이라 적힌 op 이 몰래 파일을 쓰는 일을 막는다(15장).

규칙 2 — 순수성. `fn` 이 호출자의 버퍼에 쓰면 `E-EFFECT-PURITY` 다. `fn` 은 값을 계산하고 `proc` 은 일을 한다. 판정의 기준이 **낱말의 모양**이면 구멍이 난다. 한때 액터의 핸들러가 `proc` 이 아니라는 이유로 순수로 세어져, 액터 상태를 고치면서 `effects none` 을 선언해도 아무도 반대하지 않았다. 같은 죄인데 한쪽은 잡히고 한쪽은 안 잡혔다.

규칙 3 — `effects` 절은 원자의 집합이다.

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
```

실행 결과

```
$ 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
```

실행 결과

```
$ 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
```

같은 원자를 두 번 적으면 거절된다. 집합에 중복은 뜻이 없고, 읽는 사람이 그것이 잡음인지 판단해야 한다. 없는 원자는 오타로 잡힌다 — 진단이 말하듯 오타가 조용히 지나가면 그 op 은 **순수하다고 선언한 것**이 된다. 당연해 보이는 규칙이지만, 구현이 절의 **첫 낱말 하나만** 읽던 때가 있었다. 그때 `effects io alloc` 은 `io` 로만 등록되었고 둘째 원자부터는 적어도 없는 것과 같았다. 선언을 읽는 코드의 결함은 선언 체계 전체를 무력하게 만든다.

규칙 4 — 멈춤은 효과가 아니다. 이 장에서 가장 생각할 거리다.

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 범위를 벗어나면 멈추지만, 이 op 은 여전히 순수한 fn 이다
fn pick input s slice u8 . input i u64 . output u8 . do
  return index s i .
end
```

실행 결과

```
$ 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` 는 범위를 벗어나면 멈춘다. 그러면 `pick` 은 `panic` 효과를 가져야 하는가? **아니다.** `panic` 은 명시적인 `panic "…"` 만 센다. 멈출 가능성은 **계약**이 말하는 것이지 효과가 말하는 것이 아니다. `index` 의 위험은 `requires lt i (len s)` 로 **없앨 수 있고** 실제로 처리기가 그 검사를 지운다(41장). 효과 표시는 지워지지 않는다. 없앨 수 있는 것을 효과로 표시하면 그 표시는 영원히 남고, 거의 모든 op 이 `panic` 을 갖게 되어 표시가 뜻을 잃는다.

```
op 을 끝내는 것          어디에 적나              지워지나
panic "…"  (의도)        effects panic            아니다 – 늘 남는다
범위 밖 index (위반)     requires lt i (len s)    증명되면 검사가 지워진다
```

> **흔한 오해. 멈출 수 있는 op 은 순수하지 않다**
>
> 효과는 "이 op 이 무엇을 하는가" 이고 계약은 "언제 안전한가" 다. 둘을 섞지 않았기 때문에 `panic` 표시가 여전히 정보를 가진다. `panic` 이 붙어 있으면 그 op 은 **의도적으로** 프로그램을 끝낼 수 있다는 뜻이다. 범위 밖 색인은 의도가 아니라 계약 위반이고, 계약이 그 자리를 맡는다.

44.3 권한 — 효과의 짝

효과가 "무슨 종류의 일을 하나" 라면, 권한은 "그 일을 할 **권리**를 어디서 받았나" 다. `cap io` 를 인자로 받아야 화면에 쓸 수 있고, 아무 데서나 전역으로 꺼내 쓰지 못한다. 이 객체-권한 모형이 두 가지를 준다.

- **감쇠(attenuation).** 받은 권리를 좁혀서 넘길 수 있다. 평면 효과 격자로 못 하는 정밀도를 여기서 얻는다.
- **국소 감사.** op 의 서명만 보면 무엇에 손댈 수 있는지 **전부** 보인다. 숨은 전역 통로가 없다는 것이 요점이다.

주는 것	어떻게
최적화의 정당성	`effects none` 이면 차례 바꾸기·공통 부분식 제거·지우기가 관측 동작을 바꾸지 않는다
감사할 수 있음	`io` 를 하는 코드를 선언에서 찾을 수 있다
등급과 프로파일 게이팅	작은 기계에서 `alloc` 하는 op 을 번역 오류로 막는다
놀람이 없음	순수하다고 적힌 것이 정말 순수하다

표 44.2 — 효과 체계가 주는 안전

44.4 효과 건전성 — 순환이 아니다

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
```

실행 결과

```
$ lowentc --run main recursive.low
*****
main() = 6
```

`stars` 는 자기를 부르는 재귀이고 `effects io` 를 선언한다. 처리기는 호출 자리에서 피호출자의 **본문**을 다시 걷지 않고 **선언**을 쓴다. 언뜻 순환처럼 보인다 — 선언이 선언을 믿는다.

> **수학. 효과 건전성(`LowentEffect.v` 의 `effect_sound`)**
>
> `prog_ok P = true -> lookup P f = Some (d, b) -> performs P (ECall f) a -> In a d`. 검사를 통과한 프로그램에서 op `f` 를 부르면, 실제로 일어나는 모든 효과는 `f` 가 선언한 것 안에 있다. 호출 깊이와 무관하다. 순환이 아닌 까닭은 귀납이 **프로그램**이 아니라 **실행**에 대한 것이기 때문이다. 효과가 하나 일어나면 그것은 어떤 유한한 호출 깊이에서 일어나고, 그 깊이에 대한 귀납이 닫힌다. 그래서 재귀도, 상호 재귀도, 끝나지 않는 재귀도 따로 다룰 것이 없다. 자기를 부르는 op 이 검사를 통과하고, `io` 가 실제로 일어나며, `alloc` 은 **일어날 수 없다**는 예제가 증명 파일에 함께 있다.

44.5 효과에 기댄 최적화는 적법하다

효과 건전성 위에 최적화의 적법성이 선다. 효과 층에서는 순수한 조각의 자리를 바꿔도(`pure_calls_commute`), 지워도(`dropping_pure_is_legal`) 일어나는 효과가 같다. 값 층까지 올린 증명이 `LowentOpt.v` 다. 값·상태·출력 열을 가진 모델에서 차례 바꾸기, 공통 부분식 제거, 결과 기억, 죽은 코드 제거가 넷 다 적법함을 증명했다.

열쇠는 한 줄이다 — **값은 읽는 칸에만 달렸다**(`value_depends_only_on_reads`). 그리고 전제가 장식이 아님을 반례로 보였다. 효과 있는 조각의 차례를 바꾸면 값이 14 에서 7 로 **바뀌고**, 출력하는 조각을 지우면 출력이 준다. 정리 하나에 반례 하나 — 반례가 없으면 조건이 과하게 엄격해도 정리는 참이다.

> **문. 예약된 낱말을 매개변수 이름으로 쓰면 왜 효과 진단이 아니라 이름 진단이 나오는가?**
>
> **답.** 한때는 반대였다. 매개변수 이름을 원시 포인터 낱말인 `raw` 로 쓰면 효과 추론이 조용히 오염되어 "순수하다고 선언했는데 효과를 수행한다" (`E-EFFECT-CALC`)가 떴다. 진짜 까닭은 "예약된 낱말을 이름으로 썼다" 인데 진단은 순수성을 가리켰다. 프로그래머를 엉뚱한 곳으로 보내는 진단이 가장 나쁘다. 그래서 그 낱말들을 `E-NAME-BUILTIN` 으로 넘겨 같은 죄에 진단 하나만 남겼다. 증명이 정하는 것은 무엇을 거절하나이고, 사람이 만나는 것은 왜 거절했나다 — 후자가 틀리면 전자의 값이 절반으로 준다.

44.6 증명하지 않은 것

- **구현이 모델과 같다**는 것은 시험과 두 백엔드 대조가 받치는 경험적 대응이다(50장).
- `task_group` 이 `concurrent` 를 **흡수하는** 규칙은 ⊆ 규율이 아니라 요구의 해소라 모델 밖이다.
- 효과 기반 최적화에서 **호출 경계**(순수 판정을 op 경계로 올리는 일)와 **멈추는 시점**은 밖이다.
- **평면 격자의 정밀도 한계.** `io ⊑ {read, write}` 같은 정련은 하지 않았고, 그것을 권한 감쇠로 대신하는데 둘이 같은 정밀도를 준다는 논증은 없다.
- **effect-row 다형성이 없다.** "호출자의 효과를 그대로 물려받는 op" 을 쓸 수 없어, 고차 op 은 효과를 가장 넓게 선언해야 한다.

- **일부 원자는 어휘만 있다.** `page_fault·blocking·cancel·detach` 따위는 선언할 수 있어도 그것을 일으키는 연산이 아직 없어, 그 원자의 전파는 실전 검증이 얇다.

> **복습 정리**
>
> 효과는 원자의 집합이고 부분집합 순서로 격자를 이루며, 전파는 합집합, 게이팅은 부분집합 판정이다. 처리기는 전파 · 순수성 · 원자의 집합(중복과 오타 거절) · 멈춤은 효과가 아님의 네 규칙을 강제하고, 권한이 효과의 짝으로 감쇠와 국소 감사를 준다. 효과 건전성은 실행에 대한 귀납이라 재귀가 있어도 순환이 아니며, 그 위에서 차례 바꾸기·공통 부분식 제거·기억·죽은 코드 제거가 반례와 함께 적법함이 증명되었다.

45 경합과 병렬의 증명 — 규율이 메모리 모델을 대신한다

이 장의 차례

45.1 Bernstein 조건 — 1966 년의 답 490
45.2 독립이면 경합이 없다, 그리고 그 조건이 필요하다 490
45.3 액터 사이에는 반드시 메시지가 있다 491
45.4 가둠 — 무거운 도구가 필요한 곳 491
45.5 경합이 없는 것과 결정적인 것은 다르다 491
45.6 증명하지 않은 것 492

먼저 알아야 할 것

25장 액터 · 상태는 액터 안에 잠기고 소유는 메시지로 옮겨 간다
27장 병렬 되풀이와 원자 연산 · 나눌 수 있는 되풀이의 세 조건과 `reduce`
42장 소유와 차용의 증명 · 한 흐름에서 쓰는 이름은 하나다

돌아보기

27장에서 `W-PAR-OK` 알림은 VM 이 순차로 도는 것이 왜 **올바른 구현**이라고 했는가?

> **답.** 나눌 수 있는 조건을 만족하면 병렬 결과가 순차 결과와 비트까지 같다는 정리가 증명되어 있기 때문이라고 했다. 이 장은 그 정리와, 그보다 먼저 서야 하는 "데이터 경합이 없다" 는 정리를 다룬다. 둘은 같은 전제에서 나오는 다른 결론이다.

이 장의 필요성과 맥락

두 흐름이 같은 자리를 동시에 만지고 그중 하나가 쓰기면 데이터 경합이다. C·C++ 에서는 정의되지 않은 동작이다. 경합을 다루는 보통의 길은 기억 차례 같은 메모리 모델을 배우는 것인데, 어렵다. 이 언어의 주장은 다르다 — **나누어 도는 되풀이(level 1)와 액터(level 2)의 코드는 메모리 모델을 볼 일이 없다.** 경합이 규율로 원천 차단되기 때문이다. 그리고 경합이 없어도 답이 매번 같다는 보장은 따로 필요하다. 이 장은 두 주장을 모두 무거운 동시성 논리 없이 순수 Coq 으로 세운 이야기다. 명제를 정확히 읽으면 필요한 도구가 준다.

이 장이 끝나면

Bernstein 독립이라는 세 줄의 집합 조건과 경합의 정의를 알게 된다. level 1 에서 독립이면 경합이 없고 그 조건이 필요하다는 정리 짝, level 2 에서 서로 다른 액터의 접근 사이에 메시지가 반드시 있다는 정리를 익힌다. 경합이 없는 것과 결정적인 것이 왜 다른지, 병렬 결과가 순차와 비트까지 같다는 DET-1, 스케줄러의 자유를 주는 DET-2, 결합법칙이 축약 나무의 모양을 자유롭게 하는 DET-3 과 각각의 반례도 보게 된다.

이 장에서 답할 질문

1 "대체로 같은 답" 이면 되지 않는가?

45.1 Bernstein 조건 — 1966 년의 답

두 태스크의 **읽는 자리 집합**과 **쓰는 자리 집합**을 각각 `R`·`W` 라 하자. 두 태스크가 **독립**이라는 것은 세 줄이다.

```
W₁ ∩ W₂ = ∅      둘이 같은 자리에 쓰지 않는다
W₁ ∩ R₂ = ∅      하나가 쓰는 자리를 다른 하나가 읽지 않는다
R₁ ∩ W₂ = ∅      그 반대도 아니다
```

교집합이 비었다는 세 줄이 전부다. `R₁ ∩ R₂` 는 조건에 없다 — 같은 자리를 함께 **읽는 것**은 괜찮다. 경합은 "두 태스크가 같은 자리를 건드리고 적어도 하나는 쓰기" 로 정의한다. 보통의 정의에는 "두 접근이 앞뒤로 정해지지 않았다" 가 붙는데, level 1 에서는 그 조건이 저절로 만족된다. 흐름이 갈라진 뒤 모이기 전까지 태스크들 사이에는 앞뒤 간선이 아예 없기 때문이다.

처리기의 진단이 이 조건을 그대로 인용한다.

examples/ch45/par_overlap.low

```
module par_overlap .
rem expect: E-PAR-WRITE

rem 모든 조각이 첫 칸에 쓴다 --- 남는 값이 누가 마지막이었는지에 달린다
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
```

실행 결과

```
$ 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 = ∅)
```

모든 조각이 첫 칸에 쓰므로 `wr ∩ wr = ∅` 이 깨졌고, 다른 조각의 칸을 읽으므로 `rd ∩ wr = ∅` 도 깨졌다. 27장의 세 조건 — 자기 몫만 읽기, 자기 몫만 쓰기, 되풀이를 넘어 사는 자리에 쓰지 않기 — 이 곧 Bernstein 조건이다.

45.2 독립이면 경합이 없다, 그리고 그 조건이 필요하다

> **수학. level 1 — 나누어 도는 되풀이(`LowentDRF.v`)**
>
> `l1_no_race : forall t1 t2 l, indep t1 t2 -> ~ races t1 t2 l.` 두 태스크가 Bernstein 독립이면 **어떤 자리에서도** 경합하지 않는다. 증명은 다섯 줄이다. 경합이 있다고 가정하면 "하나가 쓴다" 는데, 독립 조건이 그 자리에 대한 상대의 접근을 세 경우 모두 부정한다. 모순이다.

> `l1_condition_is_necessary : ~ indep writer reader /\ races writer reader 0`. 조건을 어기는 짝 — 하나는 자리 0 에 쓰고 하나는 자리 0 을 읽는다 — 을 실제로 만들면 경합이 **있다**.

둘째 정리가 정직한 증명의 표식이다. "조건을 만족하면 안전하다" 만 증명하면, 조건이 과하게 엄격해도 정리는 참이다 — 아무것도 통과시키지 않는 조건도 안전하다. 그래서 **반례를 함께** 증명한다. 조건을 느슨하게 하면 정말로 깨진다는 것. 정리 하나에 반례 하나 — 이 부가 여러 곳에서 쓰는 방식이다.

경합이 없으면 널리 알려진 DRF-SC 정리(Adve-Hill)가 발동해 프로그램이 순차 일관성처럼 행동한다. 곧 순서를 뒤죽박죽으로 상상할 필요가 없다.

45.3 액터 사이에는 반드시 메시지가 있다

> **수학. level 2 — 액터(`l2_accesses_are_separated_by_a_message`)**
>
> 실행 흔적에서 **서로 다른** 액터가 같은 자리 `l` 을 접근한다면, 그 두 접근 **사이에** 그 자리의 소유권을 넘기는 메시지가 반드시 있다.

논증의 사슬은 이렇다. 한 자리는 정확히 한 액터가 소유한다. 소유자만 접근한다. 소유권은 메시지로만 옮겨진다. 그런데 두 접근의 액터가 다르다. 따라서 그 사이에 소유권 이전이 있었다. 메시지는 앞뒤를 정하는 간선이고, 앞뒤가 정해진 두 접근은 **정의상 경합이 아니다**. 액터 모형에서 가장 흔한 실수 — 참조를 메시지로 보내고 보낸 쪽도 계속 쓰는 것 — 는 소유권 이전이라 보낸 쪽이 그 자리를 잃으므로 막힌다(25장의 `handoff`).

42장과의 연결이 요점이다. 한 흐름에서 "쓰는 이름은 하나" 를 강제해 두면 그 규칙이 그대로 "두 태스크가 겹치지 않음" 이 된다. 안전 규칙 하나가 두 곳에서 값을 낸다.

45.4 가둠 — 무거운 도구가 필요한 곳

```
level 1 (나누어 도는 되풀이) ─┐
level 2 (액터 격리)           ├→ 경합이 규율로 없다 → DRF-SC → 순서대로 생각한다
level 3 (원자 연산 · 락)      └→ 경합이 실제로 있다 → 메모리 모델이 필요하다
```

> **흔한 오해. 동시성을 증명하려면 반드시 분리논리 같은 무거운 도구가 필요하다**
>
> 처음 계획은 "안전 코드에 경합이 없다" 를 스케치로 두고 기계 증명은 무거운 동시성 논리로 하는 것이었다. 다시 보니 그 명제는 메모리 모델에 대한 주장이 아니었다. "규율이 경합을 막는다" 는 **집합론**의 주장이다. 무거운 도구가 정말 필요한 곳은 level 3 뿐이고, 증명 파일 가운데 Iris 를 쓴 것은 락 하나다(47장). "동시성이니까 분리논리" 는 반사적 판단이었다.

45.5 경합이 없는 것과 결정적인 것은 다르다

같은 입력으로 두 번 돌렸는데 답이 다를 수 있다. 경합이 없어도 그렇다. 부동소수 덧셈을 여러 코어가 나눠 더한 뒤 합치면, 합치는 **순서**가 달라질 때 마지막 비트가 달라진다. 각자 자기 몫만 만졌으니 경합은 없다. 그런데 답이 다르다.

이 언어의 주장은 강하다. **level 1 병렬의 결과는 순차 실행 결과와 비트까지 같다.** 근사도, 오차 허용도 없다. 그 주장을 세우는 조건이 정확히 셋임을 증명했다(`LowentPar.v`). 결정성은 경합의 문제가 아니라 Bernstein 조건의 문제다. 같은 조건이 다른 결론을 준다.

조건	있으면	없으면
DET-1 겹치지 않게 나누기	병렬 = 순차(`det1_par_eq_seq`)	차례가 결과를 바꾼다 (`overlap_is_nondeterministic`)
DET-2 의존 간선 지키기	나머지 차례는 자유 (`det2_schedule_free`)	— 자유를 안 쓰면 손해만 본다
DET-3 모으는 연산의 결합법칙	나무 모양이 결과를 바꾸지 않는다 (`assoc_shape_free`)	같은 잎에서 다른 결과 (`nonassoc_shape_matters`)

표 45.1 — 결정적 병렬의 세 조건 — 있으면 무엇이, 없으면 무엇이

수학. DET-1 — 병렬은 순차와 같다

태스크들이 쌍쌍이 독립이고 각자 **국소적**(자기 읽기 집합만 보고 값을 정한다)이면, 병렬 실행의 **모든 자리** `i` 의 값이 순차 실행과 같다. 따름정리의 이름이 주장 그대로다 — `det1_bit_identical`. 증명의 갈림길이 교육적이다. 자리 `i` 를 첫 태스크가 쓰면, 나머지 누구도 `i` 를 쓰지 않으므로 답은 첫 태스크의 것이다. 첫 태스크가 `i` 를 쓰지 않으면, 나머지 가운데 유일한 기록자가 답을 내는데, 그 태스크의 **읽기**가 첫 태스크의 쓰기와 겹치지 않으므로 원래 상태를 읽든 첫 태스크가 손댄 상태를 읽든 **같은 값**이다. 둘째 경우가 "국소적" 이 왜 필요한지 말한다.

반례는 두 태스크가 같은 자리에 각각 1 과 2 를 쓰는 것이다. 차례에 따라 답이 2 아니면 1 이다.

DET-2 는 충돌하지 않는 두 태스크의 차례를 바꿔도 결과가 같다는 정리다. 증명은 **인접 교환에 대한 귀납**이다. 인접한 둘을 바꿀 수 있으면, 그것을 되풀이해 "충돌 순서를 지키는 임의의 재배열" 에 닿는다 — 정렬 알고리즘과 같은 발상이다. 스케줄러에게 최대한의 자유를 주면서 결정성을 잃지 않으니, 자유가 곧 성능이다.

DET-3 의 반례는 뺄셈이다. `(5 - 3) - 1 = 1` 이고 `5 - (3 - 1) = 3` 이다. 연산이 결합적이 아니면 언어가 나무 모양을 고정해야 하고, 스케줄러가 코어 수에 따라 나무를 바꾸면 결과가 스케줄에 달린다. 증명은 IEEE 부동소수를 공리로 들여오지 않고, 뺄셈이라는 비결합 연산의 대표로 "비결합이면 모양이 문제다" 를 일반적으로 보였다. 그래서 `reduce acc sub` 는 `E-PAR-ASSOC` 로, 부동소수 덧셈은 `E-PAR-FLOAT` 로 거절된다 (27장).

문. "대체로 같은 답" 이면 되지 않는가?

답. 값이 거의 같으니 괜찮다고 넘기면 **회귀 시험을 비트 비교로 쓸 수 없게 된다.** 재현성을 잃으면 디버깅 도구 하나를 잃는다. 병렬 합산이 코어 수마다 다른 답을 내는 것은 과학 계산에서 재현이 안 되는 가장 흔한 원인이다. 이 언어에서 "대체로" 는 없다 — 세 조건을 확인할 수 없으면 번역이 거절한다.

45.6 증명하지 않은 것

- **DRF-SC 정리 자체는 인용이다.** "경합이 없으면 순차 일관성처럼 행동한다" 는 널리 알려진 정리이고, 여기서 증명한 것은 그 정리의 **전제**(경합 없음)를 우리 규율이 만족한다는 것이다.
- **정리는 구현을 검증하지 않는다.** level 1 병렬은 구현되었지만, 정리가 못 박은 것은 **구현이 지켜야 할 조건**이다. 구현이 그 조건을 지키는지는 두 백엔드 대조가 경험적으로 받친다.
- **모델은 정적인 읽기·쓰기 집합을 가정한다.** 포인터가 섞이면 그 집합을 미리 알기 어렵고, 실제로는 타입 체계가 그것을 보장해야 한다. "국소적" 도 가정이지 검사가 아니다.
- **상태 모델은 "자리 → 값" 함수다.** 정렬·별칭·부분 쓰기 같은 실제 메모리를 모델링하지 않았다. 부동소수 자체(반올림·NaN·−0)도 다루지 않았다.

복습 정리

Bernstein 독립은 세 줄의 빈 교집합이고, level 1 에서 독립이면 경합이 없으며 그 조건이 필요하다는 것이 반례와 함께 증명되었다. level 2 에서는 서로 다른 액터의 접근 사이에 소유권 메시지가 반드시 있어 경합이 정의상 없다. 무거운 도구는 level 3 에만 필요하다. 경합이 없는 것과 결정적인 것은 다르며, 겹치지 않게 나누기 · 의존 간선 지키기 · 결합적 모으기의 세 조건이 병렬 결과를 순차와 비트까지 같게 한다. 각 조건에는 빠지면 무엇이 깨지는지 보이는 반례가 붙어 있다.

46 약한 메모리의 증명 — 기본값이면 순서대로 생각해도 된다

이 장의 차례

46.1 순차 일관성과 그 반례 495
46.2 실행은 그래프다 495
46.3 기본값이면 나올 수 없고, relaxed 면 나올 수 있다 495
46.4 막는 것은 정확히 그 공리다 496
46.5 메시지 전달 — 약한 것은 이 모양을 위해 있다 496
46.6 모든 실행에 대해 497
46.7 약하게 적으면 거동이 늘어난다 497
46.8 문헌의 시험 일곱 498
46.9 증명하지 않은 것 498

먼저 알아야 할 것

27장 병렬 되풀이와 원자 연산 · 기억 차례 order 다섯과 기본값 seq_cst
39장 수학 도구상자 · 부분 순서 — 비교할 수 없는 쌍을 허락하는 순서
45장 경합과 병렬의 증명 · level 1·2 는 경합을 규율로 없앤다

돌아보기

45장의 가둠 그림에서 메모리 모델이 필요한 곳은 어디였는가?

> **답.** level 3 — 원자 연산과 락 — 이었다. 그곳에서는 두 흐름이 같은 자리를 실제로 두드리므로 규율이 경합을 없애 주지 못한다. 이 장은 그 자리에서 "무엇을 언제 볼 수 있는가" 를 정하는 약한 메모리 모델을 정면으로 본다.

이 장의 필요성과 맥락

CPU 와 컴파일러는 명령의 차례를 바꾼다. 흐름이 하나면 티가 나지 않지만, 여러 흐름이 같은 메모리를 보면 티가 난다 — 내가 쓴 차례대로 남이 보지 않는다. Lowent 는 원자 연산에 다섯 가지 세기를 드러내고(relaxed < acquire·release < acq_rel < seq_cst), 기본을 가장 센 seq_cst 로 둔다. 약한 것은 적어야 얻고, 그 코드는 감사 대상이다. 이 결정은 두 명제에 기댄다. 기본값이면 순서대로 생각해도 된다는 것, 그리고 약한 것은 **정말로** 그것을 깬다는 것. 하나만 증명하면 결정이 정당하지 않다. 첫째가 없으면 "기본 seq_cst" 는 위안일 뿐이고, 둘째가 없으면 "적고 감사한다" 가 과장이 된다.

이 장이 끝나면

순차 일관성이 무엇인지와 그것을 깨는 표준 반례(저장 버퍼링), 실행을 간선 여러 종류의 그래프로 보고 순환이 있으면 그 실행이 존재하지 않는다고 판정하는 RC11 모델을 알게 된다. 기본값이면 반례가 나올 수 없고 relaxed 면 나올 수 있다는 정리 짝, 막는 것이 정확히 어느 공리인지, release·acquire 가 메시지 전달을 실제로 동기화한다는 정리를 익힌다. 모든 실행에 대한 일반 정리, 약하게 적으면 거동이 늘어난다는 단조성, 그 증명이 찾아낸 모델의 결함도 보게 된다.

이 장에서 답할 질문

1 `order` 를 적지 않은 원자 연산은 어떻게 되는가?

46.1 순차 일관성과 그 반례

순차 일관성(sequential consistency, SC)은 "모든 흐름의 모든 접근을 **한 줄로 세울 수 있고**, 각 흐름 안의 차례가 그 줄에서 지켜진다" 는 뜻이다. 사람이 자연스럽게 상상하는 그림이 이것이다. 실제 하드웨어는 이렇게 동작하지 **않는다**.

```
처음: x = 0, y = 0

흐름 1:  x ← 1        흐름 2:  y ← 1
         r1 ← y                 r2 ← x

나쁜 결과:  r1 = 0  이고  r2 = 0
```

저장 버퍼링(store buffering)이라 부르는 표준 반례다. 왜 나쁜가? 한 줄로 세울 수 없다. `r1 = 0` 이면 흐름 1 의 읽기가 흐름 2 의 쓰기보다 먼저이고, `r2 = 0` 이면 흐름 2 의 읽기가 흐름 1 의 쓰기보다 먼저다. 그런데 각 흐름 안에서는 쓰기가 읽기보다 먼저다. 모순이다. 그런데 **실제 x86 에서 이 결과가 나온다.**

46.2 실행은 그래프다

기준 모델은 RC11(Lahav 외, PLDI 2017)이다. 실행을 그래프로 본다. 사건이 꼭짓점이고 간선이 여러 종류다.

간선	뜻
`hb`(happens-before)	차례가 보장된다
`rf`(reads-from)	이 읽기는 저 쓰기의 값을 읽었다
`mo`(modification order)	한 자리에 대한 쓰기들의 전역 차례
`fr`(from-read)	이 읽기는 저 쓰기보다 **먼저**다 — 더 낡은 값을 읽었으므로

표 46.1 — RC11 의 간선

판정은 한 문장이다. **어떤 축에서 순환이 생기면 그 실행은 존재하지 않는다.** 이것이 이 장에 필요한 수학의 전부다 — 방향 그래프에 순환이 있는가. 위상 정렬을 배웠다면 아는 개념이다. RC11 이 어려운 것은 개념이 아니라 간선 종류가 많아서다.

46.3 기본값이면 나올 수 없고, `relaxed` 면 나올 수 있다

수학. E1 — `sb_sc_impossible`

`consistent (SB SC) rf_00 mo_SB = false`. 모든 접근이 `seq_cst` 면 "둘 다 0 을 읽는" 결과는 일관성 공리를 어긴다 — 그런 실행은 없다. 순환을 직접 보자. `Wx1 –hb→ Ry0 –fr→ Wy1 –hb→ Rx0 –fr→ Wx1`. `fr` 을 읽는 법은 이렇다 — `Ry0` 은 `y` 의 처음 값을 읽었는데 `Wy1` 이 그것을 덮어썼으므로 `Ry0` 이 `Wy1` 보다 먼저다. 네 간선이 닫힌 고리를 이루므로 이 실행은 존재할 수 없다.

> **수학. E2 — `sb_relaxed_is_consistent`**
>
> `consistent (SB Rlx) rf_00 mo_SB = true`. `relaxed` 면 그 나쁜 결과가 **모든 공리를 만족한다** — 실제로 일어날 수 있다. 순차 일관성 공리는 `seq_cst` 사건에 대해서만 말하므로, `relaxed` 뿐이면 그 공리는 공허하게 참이다. 그것이 `relaxed` 의 값이자 위험이다. 제약을 걷어 내니 빠르지만, 걷어 낸 것이 바로 안전이다.

E1 이 "기본 `seq_cst`" 라는 문장을 위안이 아니라 **정리**로 만든다. `seq_cst` 만 쓰는 코드는 한 줄로 세워 상상해도 된다.

46.4 막는 것은 정확히 그 공리다

> **수학. E3 — 어느 축이 일하는가**
>
> `coherence_alone_does_not_forbid_sb` — `seq_cst` 여도 coherence 축만으로는 저장 버퍼링이 막히지 않는다. `it_is_the_sc_axiom_that_forbids_it` — 막는 것은 순차 일관성 축이다.

> **실제 사례. 모델이 틀렸음을 기계가 가르쳐 주었다**
>
> 처음 쓴 모델은 저장 버퍼링을 `seq_cst` 에서도 **통과시켰다**. 공리가 모자랐던 것이다. 그것을 증명기가 드러냈고, 공리를 보태 고쳤다. E3 의 두 정리는 그 교훈을 고정한다. 어느 축이 일하는지 적어 두면, 나중에 누가 공리를 지웠을 때 이 정리가 깨진다. 증명 도구의 진짜 값은 정리를 얻는 것보다 여기에 있다.

46.5 메시지 전달 — 약한 것은 이 모양을 위해 있다

E1 E3 은 "약한 것이 위험하다" 였다. 그러면 약한 것은 왜 있는가? 정확히 이 모양을 위해서다.

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
```

실행 결과

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

데이터를 먼저 쓰고(relaxed), 깃발을 release 로 쓴다. 다른 흐름은 깃발을 acquire 로 읽고, 1 이면 데이터를 읽는다. 나쁜 결과는 "깃발 1 을 읽고도 데이터 0 을 읽는" 낡은 값이다.

> **수학. E4 — 메시지 전달**
>
> mp_relacq_forbids_stale — release·acquire 면 낡은 값을 보는 것이 **불가능**하다. mp_relaxed_allows_stale — 둘 다 relaxed 면 **가능**하다. mp_sync_is_what_forbids_it — 차이를 만드는 것은 release 쓰기와 acquire 읽기 **사이에** 서는 동기화 간선 sw 다. 논증은 이렇다. sw 가 데이터 쓰기에서 데이터 읽기로 가는 hb 를 만든다. 그런데 데이터 읽기가 처음 값을 읽었고 데이터 쓰기가 그보다 mo 로 나중이므로 반대 방향 fr 이 선다. 같은 자리에서 순환이 생기므로 coherence 축이 막는다.

E4 는 E1 과 짝이다. E1 은 "기본값이 안전하다", E4 는 "적으면 동기화된다". 설계 결정의 양쪽이 모두 기계 위에 있다.

> **수학. E5 — sb_relaxed_has_a_race**
>
> 저장 버퍼링의 (0, 0) 결과가 나오는 실행은 **경합을 담고 있다**. 그런데 45장에서 보았듯 level 1·2 는 그런 실행을 만들 수 없다. 그러므로 level 1·2 코드는 이 장을 읽지 않아도 된다 — 약한 메모리는 level 3 만의 문제이고, 가둠이 성립한다.

46.6 모든 실행에 대해

위의 정리들은 대부분 vm_compute. reflexivity. 로 끝난다. consistent 를 참·거짓을 내는 판정 함수로 정의했으므로, 특정 실행에 대해 그 함수를 계산해 버리면 증명이 끝난다. 짧고, 모델을 고치면 저절로 다시 확인된다. 대가는 **특정 실행**에 대한 결과라는 것이다. 설계 결정이 걸린 것이 바로 그 표준 반례들이므로 결정을 정당화하기에는 충분하지만, 거기서 멈추지 않았다.

> **수학. 모든 접근이 seq_cst 면 순차 일관적이다(LowentRC11SC.v 의 all_sc_is_sc)**
>
> all_sc E = true -> consistent E rf mo = true -> sc_com_acyclic E rf mo = true. 모든 접근이 seq_cst 이고 그 실행이 RC11 일관적이면 po ∪ rf ∪ mo ∪ fr 에 순환이 없다 — **모든 실행 · 모든 rf · 모든 mo** 에 대해. 공리적 메모리 모델에서 "이 실행은 순차 일관적이다" 를 그렇게 정의하는 것이 표준이다. 증명의 엔진은 한 줄이다 — **부분 관계는 비순환성을 물려받는다.**

그리고 순환이 없다는 것에서 한 걸음 더 갔다. "비순환이면 전순서가 있다" 를 추상적으로 논증하는 대신 **위상 정렬을 함수로 짓고** 그 결과가 진짜 선형 확장임을 증명했다(topo_linearises·sc_witness_orders_everything). 모든 접근을 한 줄로 늘어놓았고, 그 줄이 프로그램 차례와 통신 차례를 전부 지킨다. 좋은 저장 버퍼링 실행에서 실제로 [0; 1; 2; 4; 3; 5] 라는 줄이 나온다.

46.7 약하게 적으면 거동이 늘어난다

> **수학. 단조성(LowentRC11Mono.v 의 consistent_monotone)**
>
> 같은 실행에서 기억 차례만 약하게 한 것도 일관적이다 — 모든 실행 · 모든 rf · 모든 mo · 모든 배정에 대해. 곧 **약하게 적으면 가능한 거동이 늘어난다 (줄지 않는다).** 간선 수준에서도 약하게 하면 동기화 간선이 줄고 없던 것이 생기지 않는다(sw_weaken). sc_is_the_strongest·rlx_is_the_weakest 로 세기 순서 자체가 정리다.

뒤집으면 실무의 문장이 된다 — **기억 차례를 세게 적는 것은 언제나 안전한 방향이다.** 기본값을 `seq_cst` 로 둔 근거가 이 문장이고, 그것은 믿음이 아니라 정리다.

> **실제 사례. 단조성 증명이 찾아낸 순서의 결함**
>
> 옛 세기 순서는 `acq_rel` 을 `seq_cst` **위에** 두었다. 동기화 간선만 세고 순차 일관성 공리를 보지 않았기 때문이다. 그런데 저장 버퍼링은 `acq_rel` 에서 일관적이고 `seq_cst` 에서 일관적이지 않다. 그 순서로는 단조성이 **거짓**이었다(`old_order_is_not_a_strength_order` 가 그것을 계산한다). 유계 전수 계산이 이것을 못 본 까닭도 분명했다 — 그 골격들이 순차 일관성 축을 건드리지 않았다. 순서를 고쳐 `seq_cst` 를 유일한 꼭대기로 만들었다.

46.8 문헌의 시험 일곱

손으로 고른 실행만 쓰면 모델의 공리가 모자란 것을 못 잡는다(E3 의 실패담). 그래서 문헌의 이름 있는 시험을 먹였다(`LowentRC11Sweep.v`).

시험	`seq_cst`	`relaxed`
SB · LB · MP · 2+2W · IRIW	금지	허용
CoRR · CoWR	금지	**금지**

표 46.2 — litmus 시험 일곱의 판정

마지막 줄이 배운 것이다. **"`relaxed` = **아무 보장 없음**" 이 아니다.** 저장 버퍼링을 막는 것은 순차 일관성 축이고 CoRR·CoWR 을 막는 것은 coherence 축인데, 후자는 기억 차례와 무관하게 늘 있다.

> **문. `order` 를 적지 않은 원자 연산은 어떻게 되는가?**
>
> **답.** `seq_cst` 다. 아무 표시 없는 코드가 가장 안전하다. 기본값을 `relaxed` 로 두면 아무것도 적지 않은 코드가 위험해지고, 위험을 **요청하지 않았는데** 얻게 된다. 위험은 적어서 얻는 것이어야 한다. 그리고 한 연산에 뜻이 없는 조합 — 읽기에 `release`, 쓰기에 `acquire` — 은 번역에서 거절된다 (27장).

> **흔한 오해. 성능이 필요하면 원자 연산을 모두 `relaxed` 로 적으면 된다**
>
> E2 가 말하듯 `relaxed` 는 "둘 다 0 을 읽는" 결과를 **실제로 허락한다**. 제약을 걷어 낸 만큼 빠를 수 있지만 걷어 낸 것이 바로 순서대로 사고해도 된다는 보장이다. 이 판에서 약한 차례를 명시한 코드 가운데 증명이 있는 것은 `spsc` 하나뿐이고(47장), 나머지는 감사 대상이다. 기본값 `seq_cst` 로 쓰고 측정한 뒤, 약하게 해야 할 한 자리를 골라 그 자리의 논증을 적는다.

46.9 증명하지 않은 것

- **특정 lock-free 알고리즘을 `relaxed` 로 검증하는 프로그램 논리**는 이 장에 없다. 모델 안의 메타정리(순차 일관성 · 단조성 · 순서 확장)는 증명했고, 프로그램 논리로 검증한 것은 SPSC 링 버퍼 하나이며 그것도 빌린 증명이다(47장).
- **"비순환이면 위상 정렬이 반드시 성공한다"** 는 아직 없다. 지금은 순서를 실제로 얻고 그것이 옳음을 알며, 실패하면 `None` 으로 알 수 있다 — 조용히 틀린 순서를 내는 일은 없다.
- **모델은 손으로 옮긴 RC11 이다.** 논문의 정의와 맞는지는 사람이 읽어서 확인했고, 한 번 틀렸다(E3).
- **시험은 표본이다.** fence 를 쓰는 시험이 없고(이 모델에 fence 사건이 없다), 골격마다 rf·mo 는 나쁜 결과 하나씩이다.

- **컴파일러가 이 모델을 지키는지는 별개다.** 기억 차례는 C11 atomic 으로 1 대 1 방출되므로 나머지는 C 컴파일러의 몫이다(신뢰 기반). 그 대응과 “원자 연산의 차례를 바꾸지 않는다” 는 시험으로 확인한다.

> **복습 정리**
>
> 순차 일관성은 모든 접근을 한 줄로 세울 수 있다는 뜻이고, 저장 버퍼링이 그 표준 반례다. RC11 은 실행을 간선 네 종류의 그래프로 보고 순환이 있으면 그 실행이 없다고 판정한다. 모든 접근이 `seq_cst` 면 반례가 불가능하고 `relaxed` 면 가능하며, 막는 것은 순차 일관성 공리다. `release·acquire` 는 메시지 전달을 실제로 동기화하고, 그런 나쁜 실행에는 경합이 있어 level 1·2 와 무관하다. 모든 실행에 대해 `seq_cst` 는 순차 일관적이고, 약하게 적으면 거동이 늘어나므로 세게 적는 것이 언제나 안전한 방향이다.

47 락의 증명 — 무거운 도구가 실제로 필요한 자리

이 장의 차례

47.1 분리논리의 어휘 501
47.2 토큰은 둘일 수 없다 501
47.3 도구를 필요한 만큼만 502
47.4 rwlock — 같은 문장의 두 증명 502
47.5 교착 자유와 굶주림 없음 502
47.6 약한 메모리에서 빌린 증명 — spsc 503
47.7 증명하지 않은 것 504

먼저 알아야 할 것

26장 태스크와 채널 · 나누어 가지는 자물쇠 상태는 이 판에 아직 없다(E-LOCK-NOTYET)
34장 그릇과 정렬 · spsc 는 락 없는 단일 생산자·단일 소비자 링 버퍼다
46장 약한 메모리의 증명 · release·acquire 는 메시지 전달을 동기화한다

돌아보기

45장은 무거운 동시성 논리가 필요한 곳을 어디라고 했고, 왜 거기만이라고 했는가?

> **답.** level 3, 그중에서도 락이라고 했다. level 1·2 의 규율은 같은 자리를 두 흐름이 만지는 일을 아예 없애서 집합론이면 충분했다. 그러나 스핀락은 두 흐름이 **같은 워드를 실제로 동시에 두드리는** 장치다. 여기서는 규율이 안전을 줄 수 없고, 불변식이 지켜야 한다. 이 장이 그 추론 — 동시성 분리논리 — 을 다룬다.

이 장의 필요성과 맥락

설계는 "lock·rwlock 은 원자 CAS 위의 라이브러리다" 라고 정했다. 그 문장이 참이려면 CAS 하나로 세운 스핀락이 정말로 상호배제를 주어야 한다. 그리고 이 저장소의 증명 파일 대부분이 순수 Coq 인데 락만 Iris 를 쓰는 까닭을 이해해야 도구를 필요한 만큼만 쓴 판단을 믿을 수 있다. 이 장은 분리논리의 어휘를 처음부터 소개하고, 락·rwlock·교착·굶주림의 정리를 읽고, 약한 메모리에서 **빌린** 증명이 표준 라이브러리의 spsc 를 어떻게 받치는지 본다. 락은 이 부에서 경계가 가장 많은 주제이기도 하다.

이 장이 끝나면

"서로 다른 조각" 을 뜻하는 접속사 ∗, 명세를 적는 호어 삼중항, 여러 흐름이 나누어 가지는 불변식, 실행에는 없고 증명에만 있는 유령 토큰을 알게 된다. 토큰이 둘일 수 없다는 정리에서 시작해 락 만들기·잡기·놓기의 명세가 무엇을 막는지 익힌다. rwlock 의 두 증명, 오름차순 잠금의 교착 자유와 라운드 로빈의 굶주림 없음, 그리고 약한 메모리의 프로그램 논리를 빌려 spsc 를 확인한 일과 그 대가도 보게 된다.

이 장에서 답할 질문

1 증명된 락이 있는데 이 판에서 lock 은 왜 없는가?

47.1 분리논리의 어휘

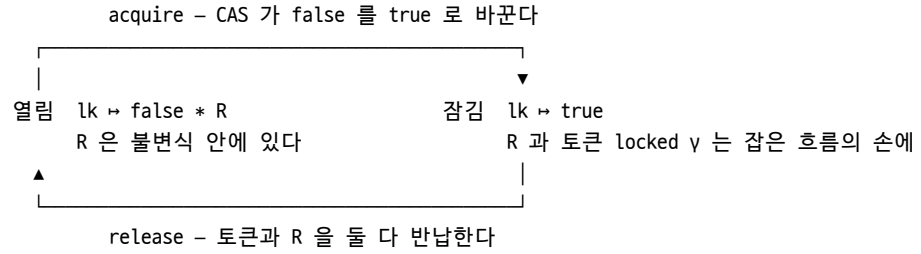
보통의 논리에서 `A ∧ B` 는 “둘 다 참” 이다. 분리논리는 접속사를 하나 더 둔다. `A ∗ B` 는 “A 와 B 가 참이고, **서로 다른 메모리 조각**에 대한 것이다” 로 읽는다. 메모리에 대한 주장이 **소유**를 함의해야 하기 때문이다. `x ↦ 3` 은 “x 가 3 을 담고 있다” 가 아니라 “나는 x 를 소유하고 그 값이 3 이다” 다. 그래서 `x ↦ 3 ∗ y ↦ 4` 는 x 와 y 가 다른 자리임을 말하고, x 에 쓰는 것이 y 에 대한 주장을 망가뜨리지 않는다. 이것이 국소 추론이다.

12장의 배타 규칙과 같은 발상이다. 실은 순서가 반대다 — Rust 의 차용 검사기가 분리논리의 발상을 타입 체계로 옮겼고, 이 언어의 배타 규칙도 그 계보에 있다. 42장은 이 장의 생각을 번역 시각으로 내린 것이다.

어휘	뜻
호어 삼중항 `{{{ P }}} e {{{ Q }}}`	`P` 를 만족하는 상태에서 `e` 를 돌리면, 끝났을 때 `Q` 가 성립한다
불변식 `lock_inv γ lk R`	`(lk ↦ false ∗ R) ∨ (lk ↦ true)` — 열려 있으면 자원 `R` 이 불변식 안에, 잠겨 있으면 잠근 흐름의 손에
유령 토큰 `locked γ`	`own γ (Excl ())` — 실행에는 없고 증명에서만 도는 회계 장치

표 47.1 — 락 증명의 어휘

불변식 한 줄이 락의 전부다. `R` 은 두 곳에 동시에 있을 수 없으므로, `R` 을 가진 흐름은 하나뿐이다.



47.2 토큰은 둘일 수 없다

> **수학.** `LowentLock.v` **의 정리들**
>
> `locked_exclusive : locked γ -∗ locked γ -∗ False.` 토큰을 둘 가졌다고 가정하면 모순이다. 이것이 상호배제의 기계적 심장이다.
>
> `newlock_spec : {{{ R }}} newlock #() {{{ lk γ, RET lk; is_lock γ lk R }}}.` 보호할 자원을 **넘겨주면** 락을 얻는다. 락을 만드는 대가로 자원을 포기한다 — 이제 그 자원에 닿는 유일한 길은 락을 잡는 것이다.
>
> `acquire_spec : {{{ is_lock γ lk R }}} acquire lk {{{ RET #(); locked γ ∗ R }}}.` 잡으면 토큰과 자원을 **둘 다** 받는다. `∗` 가 일한다 — 받은 자원은 내 것이고 누구의 것과도 겹치지 않으므로, 그 뒤로는 흐름이 하나인 것처럼 추론한다.
>
> `release_spec : {{{ is_lock γ lk R ∗ locked γ ∗ R }}} release lk {{{ RET #(); True }}}.` 놓으려면 토큰과 자원을 둘 다 반납해야 하고, 놓은 뒤에는 아무것도 남지 않는다.

두 흐름이 동시에 잡았다면 토큰이 둘이고, 그것은 거짓이다. 놓기의 명세 하나가 두 결함을 막는다. 잡지 않고 놓으면 `locked γ` 가 없어 사전조건을 못 맞추고, 놓은 뒤에도 자원을 쓰면 `R` 을 이미 반납했으므로 손에 없다.

증명들의 공통 모양은 셋이다. 불변식을 연다, CAS 를 실행한다(성공과 실패로 갈린다), 불변식을 닫는다. **원자적인 순간에만** 불변식을 열 수 있다는 것이 Iris 의 규칙이고, 그 덕에 "불변식이 잠깐 깨진 상태를 다른 흐름이 볼 수 없다" 가 보장된다.

> **수학. 실전 예 — 두 흐름의 증가(`incr_spec`·`two_threads_spec`)**
>
> 두 흐름이 동시에 `acquire ; c ← !c + 1 ; release` 를 돈다. 두 흐름이 동시에 증가시켜도 경합 없이 끝나고 프로그램은 값을 하나 낸다. 락 없는 `c ← !c + 1` — 고전적인 lost update — 은 `c ↦ n` 을 락에서 받아야만 쓸 수 있으므로 증명되지 않는다. 정직하게: 이 정리는 결과가 **정확히 2** 라고 말하지 않는다. 정확한 합계를 말하려면 유령 카운팅이 더 필요하다. 이 파일의 목표는 락이 자원을 지킨다는 것이었다.

47.3 도구를 필요한 만큼만

파일	도구	왜
차용 · 되풀이 · 경합 · 병렬 · RC11 · 블록 · 칸 · 세탁 · 수	순수 Coq	규율 · 집합론 · 격자 · 그래프 판정이면 된다
락 · rwlock 읽기 쪽	Iris	두 흐름이 같은 자리를 실제로 두드린다
약한 메모리 프로그램 · SPSC	iRC11 · gpfsl(빌림)	약한 메모리 위에서 프로그램을 검증해야 한다

표 47.2 — 증명 파일과 도구

Coq 을 고른 이유도 여기에 있다. Iris 는 Coq 전용이고 Lean 에는 성숙한 대응물이 없다. **언젠가 필요할 도구가 어디 있는지**가 증명 언어 선택을 정했다.

47.4 rwlock — 같은 문장의 두 증명

한 워드에 세 뜻을 담는 고전적 rwlock — 0 은 빔, n > 0 은 독자 n, −1 은 쓰기 잠금 — 에서 **쓰기 잠금의 배타성**을 직접 증명했다(`LowentRWLock.v` 의 `wlocked_exclusive`·`wlock_spec`·`wunlock_spec`). CAS 가 **0 에서만** 성공하므로 "독자가 있는데 쓰기가 들어간다" 가 막힌다. 실제 rwlock 결함의 대부분이 그것이다.

읽기 쪽의 **분수 소유권**은 Iris 가 이미 갖고 있었다. 라이브러리의 rwlock 인터페이스가 분수 술어 `ϕ : Qp → iProp` 로 적혀 있고 `rw_spin_lock` 이 검증된 인스턴스다. 독자는 분수 `ϕ q` 를 받고(여럿이 동시에), 쓰기와 읽기는 동시에 있을 수 없으며, 쓰기는 전부 `ϕ 1` 을 받는다. 값진 것은 "쓰기와 읽기는 배타적이다" 가 **같은 문장의 다른 증명**이라는 점이다. 우리 것은 "CAS 가 0 에서만 성공한다" 로(구현에 가깝게), 빌린 것은 유령 상태로(일반적으로, 분수까지) 보였다. 두 방법으로 만들고 맞대는 원리(31장)가 증명 층에서도 한 번 더 선다.

47.5 교착 자유와 굶주림 없음

> **수학. 교착 자유(`LowentDeadlock.v` 의 `no_deadlock`)**
>
> 모든 흐름이 잠금을 **오름차순으로만** 잡으면, 어떤 상태에서도 진행할 수 있는 흐름이 반드시 있다 — 전원이 동시에 막히지 않는다. 씨앗은 한 문장이다. **가장 큰 잠금을 원하는 흐름**을 보라. 그것이 막혔다면 그 잠금을 쥔 다른 흐름이 있고, 규율에 따라 그 흐름이 원하는 것은 더 크다 — 최대성에 모순이다. 규율을 어기면 실제로 교착이 난다는 상태도 계산으로 보였다(`violating_the_order_deadlocks`).

```
오름차순을 지킬 때 (A < B)          규율을 어길 때
흐름 1   A 를 쥠 → B 를 원함        흐름 1   A 를 쥠 → B 를 원함
흐름 2   A 를 원함 (기다림)         흐름 2   B 를 쥠 → A 를 원함
→ 흐름 1 이 B 를 잡고 진행한다      → 서로를 기다린다 (교착)
```

정직하게 — **도구가 이 규율을 강제하지 않는다.** 그래서 이것은 "이렇게 쓰면 안전하다" 이지 "컴파일러가 막아 준다" 가 아니다. 검사로 만들려면 잠금에 정적 순서를 붙여야 한다. 그리고 락 자체의 명세(잡으면 자원을 받는다)는 교착을 막지 않는다. "잡으면 받는다" 이지 "반드시 잡힌다" 가 아니기 때문이다. 락을 두 번 잡고 한 번 놓으면 둘째 잡기는 영원히 오지 않는다.

> **수학. 굶주림 없음(`LowentFair.v` 의 `no_starvation`)**
>
> 협력형 라운드로빈 스케줄러에서 준비된 태스크는 **자기 앞의 수 + 1 걸음 안에** 반드시 돈다. "언젠가" 가 아니라 유계다. 귀납의 측정치가 대기열의 길이가 아니라 **자리**인 것이 요점이다 — 회전은 길이를 보존하므로. 회전이 없으면 굶는다는 것도 증명했다.

한계는 분명하다. 협력형은 **양보를 가정한다.** `yield` 없이 끝없이 도는 태스크 앞에서 스케줄러가 할 수 있는 것은 없다. 우선순위와 차단도 이 모델 밖이다.

> **흔한 오해. 락을 쓰면 교착도 막아 준다**
>
> 락의 명세는 "잡으면 자원을 받는다" 이지 "반드시 잡힌다" 가 아니다. 교착 자유(`no_deadlock`)는 모든 흐름이 잠금을 **오름차순으로만** 잡는다는 규율 위에서만 성립하고, 이 판의 도구는 그 규율을 강제하지 않는다. 락을 두 번 잡고 한 번 놓으면 둘째 잡기는 영원히 오지 않는다. 증명이 주는 것은 "이렇게 쓰면 안전하다" 는 설계 규칙이고, 그 규칙을 지키는 일은 아직 사람의 몫이다.

47.6 약한 메모리에서 빌린 증명 — `spsc`

Iris 의 언어(`heap_lang`)는 순차 일관성 모델이다. 약한 기억 차례를 적은 코드를 증명하려면 RC11 위의 Iris 논리인 iRC11·gpfsl 이 필요하다. `LowentIRC11.v` 가 그 위에서 첫 정리를 세웠다 — `release` 쓰기와 `acquire` 읽기로 짠 메시지 전달 프로그램은 **RC11 에서** 읽는 쪽이 반드시 데이터를 본다. 순차 일관성을 가정하지 않는다. 46장의 E4 가 실행 하나에서 **프로그램 전체**로 올라갔다.

그 도구로 표준 라이브러리 모듈을 하나 세웠다.

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
```

실행 결과

```
$ lowentc --run main ring.low
main() = 42
```

생산자는 칸을 **먼저** 채우고 그다음에 색인을 `release` 로 공개한다. 소비자는 그 색인을 `acquire` 로 읽은 뒤에야 칸을 읽는다. `lowent_spsc_push_is_correct`· `lowent_spsc_pop_is_correct` 가 이 짝이 RC11 에서 옳다고(가득·빔 판정 포함) 말한다.

```
생산자                                  소비자
buf[i] ← x          (칸을 먼저 채운다)
tail ← i+1  release ──────────────────▶  tail  acquire  로 i+1 을 보면
                                        buf[i] 를 읽는다 → 반드시 x 를 본다 (RC11 에서)
```

> **실제 사례. 빌렸다는 것의 뜻**
>
> 알고리즘 수준의 증명은 gpfsl 의 `circ_buff` 이고, `mp_instance_gen_inv` 도 gpfsl 의 것이다. 이 저장소가 한 일은 "우리가 방출하는 기억 차례가 그 명세가 요구하는 바로 그것" 임을 진술하고 잇는 것이다. 그래서 신뢰 기반에 gpfsl 과 Iris 개발판이 더해진다. 다시 짓지 않은 이유는 하나다 — 같은 것을 두 벌 두면 갈린다. 그리고 그래서 **하나만** 넣었다. MPSC·MPMC·seqlock 은 빌릴 근거가 없고, 근거 없이 넣으면 표준 라이브러리가 "검증됐다고 적혀 있으나 아무도 검증하지 않은 것" 의 창고가 된다. "무겁다" 고 적어 두었던 이 작업을 실제로 재 보니 대부분이 도구 조립이었고 진짜 위험은 없었다. 재 보기 전에는 비용을 모른다.

> **문. 증명된 락이 있는데 이 판에서 `lock` 은 왜 없는가?**
>
> **답.** 증명한 것은 Iris 의 언어로 쓴 CAS 스핀락이고, 실제 런타임의 락이 그것과 같다는 것은 검증되지 않았다 — 이 파일과 구현 사이의 가장 큰 간극이다. 그리고 나누어 가지는 자물쇠 상태 타입은 아직 짓지 않았다(`E-LOCK-NOTYET`, 26장). 증명이 먼저 있고 구현이 나중에 오는 자리다. 없는 것을 받아들이고 나중에 짓는 대신 없다고 말한다.

47.7 증명하지 않은 것

- **락 모델은 순차 일관성이다.** 약한 기억 차례는 `heap_lang` 락 증명에 없다. 다만 기본 차례가 `seq_cst` 이므로(46장) 기본값으로 쓰는 level 3 코드는 그 모델 안에 있고, 약한 차례를 적은 코드만 iRC11 의 몫으로 남는다. 가둠이 여기서도 선다.
- **구현이 이 스핀락이라는 보장이 없다.**
- **`two_threads_spec` 은 정확한 합계를 말하지 않는다.**
- **교착 자유는 사람이 지키는 규율에 기댄다.** 굶주림 없음은 양보를 가정하고, 우선순위·차단은 밖이다.
- **우리 컴파일러가 `spsc` 의 그 프로그램을 낸다는 보장**은 없다. 기억 차례의 1 대 1 방출과 차례를 바꾸지 않음은 시험으로 확인한다.

- **빌린 증명은 신뢰 기반에 들어간다.** 두 확인기(Coq 8.20.1 · Rocq 9.2)가 같은 답을 내어 확인기 자체의 결함 가능성을 줄이지만, gpfsl 이 필요한 둘은 Rocq 9.2 에서만 확인된다.

> **복습 정리**
>
> 분리논리의 $*$ 는 서로 다른 조각을 뜻해 국소 추론을 주고, 락은 “열려 있으면 자원이 불변식 안에, 잠겨 있으면 잠근 흐름의 손에” 라는 불변식 한 줄과 유령 토큰으로 증명된다. 토큰은 둘일 수 없고, 잡으면 토큰과 자원을 받으며, 놓으려면 둘 다 반납한다. rwlock 의 배타성은 직접 증명과 빌린 증명 두 가지로 확인했고, 오름차순 잠금은 교착하지 않으며 라운드로빈은 유계 안에 태스크를 돌린다. 약한 메모리의 프로그램 논리를 빌려 `spsc` 를 확인했고, 그 대가로 신뢰 기반이 늘었다.

48 문법의 증명 — 어느 닫개로 닫아도 같은 나무

이 장의 차례

48.1 대수와 뜻함수 507
48.2 한 문장은 한 문장짜리 블록이다 507
48.3 닫개는 같은 일을 한다 508
48.4 흔한 실수 510
48.5 증명하지 않은 것 511

먼저 알아야 할 것

3장 겉모습 · 떨어진 마침표가 닫고, 개행은 공백이다
8장 식 · 전위 표기에는 우선순위가 없다
39장 수학 도구상자 · 집합과 관계 — "같다" 를 무엇으로 정하나

돌아보기

3장에서 `poly` 의 `return` 은 두 줄에 걸쳐 있었지만 한 폼이었다. 무엇이 그 폼을 닫았고, 개행은 무슨 일을 했는가?

> **답.** 둘째 줄 끝의 떨어진 마침표가 닫았다. 개행은 스페이스와 똑같은 공백이라 아무것도 닫지 않았다. 그래서 줄을 어디서 나누어도 뜻이 같았다. 이 장은 그런 표면의 성질이 왜 증명의 대상이 되는지를 다룬다.

이 장의 필요성과 맥락

여기까지의 증명은 **의미**였다 — 프로그램이 무엇을 하는가. 이 장은 **표면**이다 — 프로그램이 어떻게 생겼는가. 문법에 증명이 필요한 까닭은 실제로 겪은 일 때문이다. 파서가 . 로 폼을 조각내고 뒤 단계가 말없이 다시 붙이고 있었다 — 두 층이 서로 다른 문법을 믿었고, 그 차이가 결함으로 나왔다. "한 뜻에 한 표기" 라는 원칙이 겉모습의 취향으로 남지 않으려면, 두 표기가 같은 나무를 만든다는 것을 기계가 지켜야 한다.

이 장이 끝나면

항등원과 결합법칙을 갖는 모노이드와, 두 표기가 같은지를 함수로 정하는 뜻함수(denotation)를 알게 된다. 문장 하나가 문장 하나짜리 블록과 같다는 정리, 닫개 셋이 같은 나무를 만든다는 정리, 블록의 결합법칙을 예제로 확인한다. 같은 뜻이 증명돼도 철자를 두지 않은 개행 닫기, 언어에서 뺀 규칙과 함께 뺀 정리, 그리고 이 정리가 실제 파서에 대해 말하지 않는 것도 보게 된다.

이 장에서 답할 질문

1 이 정리가 있으면 실제 파서가 옳다고 믿어도 되는가?
2 C 에 우선순위 규칙이 수십 줄인데, 이 장의 증명은 왜 이렇게 짧은가?

48.1 대수와 뜻함수

문장을 이어 붙이는 일에는 두 성질이 있다. 이어 붙이는 방식이 묶는 순서와 무관하고(결합법칙), 빈 블록은 아무것도 아니다(항등원 — 덧셈에서 0 이 하는 일). 이 둘을 갖는 구조를 **모노이드**라 한다. 목록을 이어 붙이는 것이 대표적인 모노이드이고, 프로그램의 문장 나열이 정확히 그 구조다.

두 표기가 “같은지” 를 말하려면 기준이 필요하다. 각 나무를 **평평한 문장 나열**로 보내는 함수 `denote` 를 두고, 결과가 같으면 같다고 한다. `denote(do S end) = [S] = denote(S)` 이므로 둘은 같다. 이것이 이 장의 유일한 기술이다 — “같다” 의 뜻을 함수로 정하고, 등식을 계산으로 확인한다.

48.2 한 문장은 한 문장짜리 블록이다

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
```

실행 결과

```
$ 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` 는 `if` 의 몸에 문장 하나를 그대로 적었고, `blocked` 는 `do … end` 로 감쌌다. 답이 같고, `--fmt` 가 찍는 정규형도 한 글자 다르지 않다.

> **수학. 정리 G1(`LowentBlock.v` 의 `one_form_is_a_block`)**
>
> `forall s, denote (wrap s) = denote s.` 문장 하나를 블록으로 감싸도 뜻이 같다. 여기서 “뜻이 같다” 는 각 나무를 평평한 문장 나열로 보내는 함수 `denote` 의 결과가 같다는 뜻이다. “같다” 의 뜻을 함수로 정하고 등식을 계산으로 확인하는 것이 이 장의 유일한 기술이다. 따름정리 `body_is_one_thing` 은 몸 자리에 문장을 쓰든 블록을 쓰든 같다고 말한다.

이 정리는 설계 결정 하나를 정당화한다. C 는 문장과 복합문을 다른 종류로 갈라 놓았다. `if (c) x = 1;` 과 `if (c) { x = 1; }` 가 문법적으로 다른 것이고, 유명한 `goto fail;` 함정이 그 틈에서 나왔다 — 들여쓰기는 두 줄이 `if` 에 속한 것처럼 보였지만 문법은 한 줄만 속하게 했다. Lowent 는 가르지 않는다.

```
C       if (c)
            x = x + 10;
            y = 0;          ← 들여쓰기는 if 안처럼 보이지만 문법은 if 밖이다

Lowent  if gt a 1 . set x (add x 10) .        문장 하나 = 문장 하나짜리 블록 (G1)
        if gt a 1 . do
          set x (add x 10) .
          set y 0 .
        end                                   몸이 길면 end 가 끝을 적는다
```

48.3 닫개는 같은 일을 한다

> **수학. 정리 G2(`closers_agree`)**
>
> `forall c1 c2 h ops, close c1 h ops = close c2 h ops.` `.` 과 `)` 가운데 어느 것으로 닫아도 같은 나무가 나온다. 둘 다 “열려 있는 가장 안쪽 폼을 닫는다” 를 한다. `end` 는 이 목록에 없다 — `do … end` 는 짝인 괄호라서 `end` 는 자기 `do` 만 닫고, 안에 열린 폼이 남아 있으면 대신 닫지 않고 거절한다(`E-DOT-MISSING`). 블록은 결합적이어서 블록 안의 블록은 자기 자리에서 펼쳐진다(`block_assoc·nested_block_flattens`).

예제로 보면 이렇다.

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 인자를 여러 줄에 늘어놓아도 표시가 필요 없다 --- 개행은 공백이다
fn spread input x u64 . output u64 .
  requires le x 1000 .
do
  return add
    (mul x x)
    (add x
         1) .
end
```

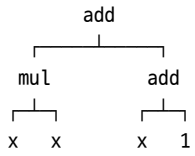
실행 결과

```
$ lowentc --run one_line closers.low 4
one_line(4) = 21
$ lowentc --run spread closers.low 4
spread(4) = 21
```

`one_line` 은 `)` 두 개가 안쪽 폼을, `.` 이 바깥 폼을 닫는다. `spread` 는 같은 폼을 네 줄에 늘어놓았는데 아무 표시가 필요 없다. 개행은 공백이기 때문이다. 두 op 의 답이 같다.

```
return add (mul x x) (add x 1) .      return add
                                          (mul x x)
                                          (add x
                                               1) .
```

둘 다 같은 나무가 된다:



개행도 닫개로 만들 수 있고 그것도 증명된다. 그래도 두지 않는다. 개행이 닫으면 줄바꿈이 의미가 되어 긴 줄을 나누는 것만으로 프로그램이 바뀐다. **같은 뜻임이 증명된다고 그 철자를 두어야 하는 것은 아니다.** 한때 쉼표·줄잇기·개행 닫기에 붙어 있던 정리들도 모두 `Qed` 였지만, 그 규칙들이 언어에서 빠지면서 정리도 함께 뺐다. 지금 `,` 와 `;` 는 `E-VOCAB-REMOVED` 로 거절된다 — 파싱되는데 아무 효과가 없는 낱말은 조용한 함정이기 때문이다. 죽은 규칙을 모형화한 정리를 남겨 두면 증명된 문서가 없는 규칙을 말하게 된다.

이 파일의 증명은 전부 계산으로 끝난다. 값은 증명의 어려움이 아니라 **명제를 골라 적었다**는 데 있다. "닫개들이 같은 일을 한다" 를 적어 두면, 누가 하나를 특별 취급했을 때 증명이 깨진다. 문법은 "여기만 예외를 두면 편한데" 로 조용히 표류하는 층이고, 정리가 그 표류를 잡는다. 막는 것은 사용자의 결함이 아니라 컴파일러의 결함이다.

> **문. 이 정리가 있으면 실제 파서가 옳다고 믿어도 되는가?**
>
> **답.** 아니다. 모델은 나무를 만드는 작은 함수들이고, 실제 파서는 그보다 훨씬 크다. 둘의 대응은 시험이 받친다. 그리고 두 백엔드 대조는 VM 과 네이티브를 비교하지만 둘은 **같은 앞단**을 지나므로, 파서가 틀리면 둘 다 똑같이 틀린다. 파서를 이 모델과 직접 대조하려던 시도는 모델이 실제 문법보다 작아서 결론을 낼 만큼의 짝을 얻지 못했다. 앞단은 이 언어의 검증에서 가장 약한 이음매다(50장).

> **문. C 에 우선순위 규칙이 수십 줄인데, 이 장의 증명은 왜 이렇게 짧은가?**
>
> **답.** 이 언어의 식은 전위 표기라 연산자 우선순위가 없다(`expr` 섬은 따로 다룬다, 8장). 우선순위가 없으니 "이 표기가 어떤 나무가 되는가" 에 답할 경우가 적다. 설계가 증명을 쉽게 만든 사례다. 그 대신 증명이 짧다고 값이 작은 것은 아니다 — 명제를 적어 두었다는 것 자체가 표류를 막는다.

> **흔한 오해. 줄을 바꾸면 문장이 끝난다**
>
> 개행은 공백이다. `closers.low` 의 `spread` 는 한 폼을 네 줄에 늘어놓았는데 아무 표시도 없이 `one_line` 과 같은 나무가 된다. 폼을 끝내는 것은 닫개(`.`·`)`) 뿐이다. 그래서 마침표를 빠뜨리면 다음 줄이 앞 폼에 이어 붙어 엉뚱한 진단이 나오고, 긴 식을 여러 줄로 나누어도 뜻이 바뀌지 않는다. 개행을 닫개로 만드는 규칙도 증명은 되지만 두지 않은 까닭이 바로 이 둘째 성질이다.

48.4 흔한 실수

> **반례. 줄 끝의 마침표를 빠뜨린다**

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 ✘ 이 줄 끝의 마침표를 빠뜨렸다
  var s u64 be add a b
  set s (mul s 2) .
  return s .
end
```

실행 결과

```
$ lowentc --check mistake_noperiod.low
mistake_noperiod.low:10:0 E-IR-ARITY: extra operands in expression (strict arity)
```

진단은 마침표를 빠뜨린 줄이 아니라 **다음 줄**을 가리킨다. 개행이 아무것도 닫지 않으니 `add a b` 뒤에 `set s (mul s 2)` 가 이어 붙어 `add` 가 인자를 넷 받은 모양이 되었고, 도구는 "인자가 남는다" 고 말한다. 줄 번호보다 한 줄 위를 먼저 본다. 이 진단이 헷갈려도 규칙은 바뀌지 않는다. 줄이 문장을 끝낸다면 긴 식을 나누는 것만으로 뜻이 바뀐다.

> **반례. C 처럼 `;` 로 닫는다**

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 ✘ C 처럼 쌍반점으로 닫는다
```

```
  return mul a 2 . ;
end
```

실행 결과

```
$ 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 `.`
```

`;` 는 예전에 마침표의 셋째 철자였다. 한 뜻에 철자가 셋이면 읽는 사람이 차이를 찾게 되므로 뺐고, 이제 `E-VOCAB-REMOVED` 로 거절한다. 조용히 받아들이지 않는 것은, 파싱은 되는데 아무 뜻이 없는 낱말이 가장 찾기 어려운 함정이기 때문이다.

48.5 증명하지 않은 것

- **실제 파서가 이 모델이라는 보장이 없다.** 모델은 나무를 만드는 작은 함수들이고 실제 파서는 훨씬 크다. 대응은 단위 시험과 회귀 시험이 받친다.
- `denote` **는 평평한 나열이다.** 위치 정보·주석·오류 복구를 모델링하지 않았다. 그러니 “같은 나무” 는 구조가 같다는 뜻이지 진단의 품질까지 같다는 뜻이 아니다.
- **어휘(토큰) 층은 별도다.** 토큰이 이 모델의 닫개 셋으로 정확히 나뉘는지는 렉서 시험의 몫이다.

복습 정리

문장 나열은 결합법칙과 항등원을 갖는 모노이드이고, 두 표기가 같은지는 평평한 나열로 보내는 뜻함수로 정해 계산으로 확인한다. 문장 하나는 문장 하나짜리 블록과 같고(G1), `.·` 는 같은 나무를 만들며(G2), 블록 안의 블록은 제자리에서 펼쳐진다. 개행 닫기는 증명돼도 두지 않았고, 언어에서 뺀 규칙의 정리는 함께 뺐다. 이 정리들은 컴파일러의 표류를 막지만, 실제 파서가 이 모델이라는 보장은 시험이 받친다.

49 해시의 증명 — 이름이 아니라 내용으로 부르기

이 장의 차례

49.1 해시 함수와 Merkle DAG 512
49.2 이름이 아니라 내용으로 부른다 513
49.3 무엇이 해시를 바꾸고 무엇이 안 바꾸나 513
49.4 증명할 수 있는 것과 없는 것 515
49.5 해시가 막는 것 516
49.6 증명하지 않은 것 516

먼저 알아야 할 것

31장 짓고 시험하기 · 빌드는 방출된 C 의 내용 해시로 캐시되고, 의존은 해시로 고정한다
3장 겉모습 · 머리의 절 차례는 하나로 고정되어 있다
48장 문법의 증명 · 같은 뜻을 가진 두 표기는 같은 나무가 된다

돌아보기

31장에서 `--lock` 은 판 번호가 같아도 무엇이 다르면 빌드를 거절한다고 했는가?

> **답.** 바이트가 다르면 거절한다고 했다. 판 번호가 같아도 내용이 바뀌었으면 다른 의존이다. 이 장은 그 "내용" 을 어떻게 적으면 **같은 프로그램**을 바이트가 아니라 뜻으로 알아볼 수 있는지, 그리고 그 적는 법에 대해 무엇을 증명할 수 있는지를 다룬다.

이 장의 필요성과 맥락

48장이 두 표기가 같은 나무를 만든다는 것이었다면, 이 장은 두 프로그램이 **같은가** 다. 캐시가 "같다" 고 판단한 두 프로그램의 뜻이 다르면 캐시는 거짓말을 한다. 반대로 뜻이 같은데 "다르다" 고 판단하면 쓸데없이 다시 짓는다. 재현 가능한 빌드, 증분 빌드, 의존 고정이 모두 이 판단에 기댄다. 해시 함수 자체는 증명할 수 없는 암호 가정이지만, 그 **앞** — 무엇을 해시에 넣고 무엇을 뺄지 — 은 증명할 수 있다.

이 장이 끝나면

해시 함수의 충돌 저항이라는 가정, 해시가 해시를 담는 Merkle DAG, 서로를 부르는 op 들을 한 덩어리로 묶어 순환을 없애는 방법을 알게 된다. `--emit-db` 의 두 해시(`iface·def`)가 주석·본문·효과 차례·계약에 어떻게 반응하는지 실제 출력으로 확인하고, 정규 인코딩의 단사성 정리가 무엇을 보증하는지, 계약 절을 정규화하지 않은 부정확함이 왜 안전한 방향인지도 이해한다.

이 장에서 답할 질문

1 `--emit-db` 의 해시가 실제 빌드 결정에 쓰이는가?

49.1 해시 함수와 Merkle DAG

해시 함수 BLAKE3-256 은 임의 길이의 바이트를 32 바이트로 줄인다. 필요한 성질은 하나다 — **다른 입력이 같은 출력을 내는 짝을 찾을 수 없다**(충돌 저항). 그래서 "해시가 같다" 를 "내용이 같다" 로 써도

된다. 이것은 수학적으로 증명된 것이 아니라 **가정**이고 신뢰 기반에 들어간다(50장). 그래도 실용적으로는 다른 어떤 가정보다 튼튼하다.

49.2 이름이 아니라 내용으로 부른다

보통의 빌드 시스템은 파일 이름과 수정 시각으로 캐시를 판단한다. 그래서 파일을 만졌지만 내용이 같으면 쓸데없이 다시 짓고(손해), 내용이 바뀌었는데 시각이 같으면 낡은 결과를 쓴다(위험). **내용 주소화**는 이름을 버리고 내용의 해시로 부른다. `lowentc --emit-db` 가 op 마다 해시 둘을 낸다 — `iface` 는 밖에서 관찰 가능한 표면(서명과 계약), `def` 는 정의 전체다. 해시 함수는 BLAKE3-256 이다.

해시는 의존을 **그들의 해시로** 품는다. 그러면 어떤 정의의 해시는 그것이 의존하는 모든 것의 내용을 반영하고, 깊은 곳의 op 하나가 바뀌면 위쪽 해시가 따라 바뀐다. 해시 자체가 의존 추적이다. 이것이 Merkle DAG 다.

서로를 부르는 op 은 순환을 만든다. `hash(A)` 가 `hash(B)` 를 담고 `hash(B)` 가 `hash(A)` 를 담으면 정의가 돈다. 그래서 순환하는 것들을 **한 덩어리** (강결합 요소, SCC)로 묶는다. 이름 없는 예비 해시로 덩어리 안의 정규 차례를 정하고, 덩어리 안의 호출은 이름 대신 그 차례의 번호를 담고, 덩어리를 한 번 해싱한 뒤, 구성원마다 (덩어리 해시, 자기 번호)를 해시로 준다. 강결합 요소로 줄이면 남는 것은 순환 없는 그래프다 — 39장의 고정점과 같은 발상이 또 나왔다.

```
A ⇄ B (서로 부른다)      C ⟶ A

1  {A, B} 를 한 덩어리(SCC)로 묶는다
2  이름 없는 예비 해시로 덩어리 안의 차례를 정한다      A = 0 · B = 1
3  덩어리 안의 호출은 이름 대신 번호를 담는다           A 가 B 를 부름 → «1 번»
4  덩어리를 한 번 해싱한다                              → H
5  구성원마다 (H, 번호) 를 해싱한다                     hash(A) = h(H, 0) · hash(B) = h(H, 1)
   C 는 hash(A) 를 담는다 – 남은 그래프에는 순환이 없다
```

49.3 무엇이 해시를 바꾸고 무엇이 안 바꾸나

examples/ch49/h1_plain.low

```
module hashed .
rem db

export fn twice input a u64 . output u64 .
do
  return add a a .
end
```

실행 결과

```
$ lowentc --emit-db h1_plain.low
fn twice/1 iface:6630705cb68be394 def:73e3a88345444202
```

examples/ch49/h1_comment.low

```
module hashed .
rem db

rem 주석을 넣어도 해시는 그대로다
```

```
export fn twice input a u64 . output u64 .
do
  rem 여기도 주석
  return add a a .
end
```

실행 결과

```
$ lowentc --emit-db h1_comment.low
fn twice/1 iface:6630705cb68be394 def:73e3a88345444202
```

H1 — 주석은 아무것도 바꾸지 않는다. 두 해시가 모두 같다. 주석을 고쳤다고 프로젝트 전체가 다시 지어지지 않는다. 그리고 "같은 프로그램" 의 정의가 바이트가 아니라 뜻이라는 것을 기계가 지킨다.

examples/ch49/h2_body.low

```
module hashed .
rem db

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

실행 결과

```
$ lowentc --emit-db h2_body.low
fn twice/1 iface:6630705cb68be394 def:64a7865b02486293
```

H2 — 본문만 바뀌면 `def` 만 바뀐다. `add a a` 를 `mul a 2` 로 바꾸자 `iface` 는 그대로이고 `def` 만 바뀌었다. 이것이 증분 빌드의 심장이다. 어떤 op 의 본문만 바뀌면 그 op 을 쓰는 쪽은 다시 컴파일하되 그 쪽의 `iface` 도 그대로이므로, 그다음 의존자는 캐시에 적중한다. 서명이 바뀌면 사슬을 따라 번지고, 본문만 바뀌면 한 단계에서 멈춘다.

```
main ──부른다──▶ util ──부른다──▶ leaf        leaf 의 본문만 바꿨다

leaf   def 바뀜 · iface 그대로
util   다시 컴파일한다 · 그 iface 도 그대로
main   캐시 적중 — 여기서 멈춘다
```

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
```

실행 결과

```
$ 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
```

실행 결과

```
$ lowentc --emit-db h3_effects_b.low
proc greet/2 iface:d864e930303f7437 def:bebc1fcf6231d282
```

H3 — 효과는 집합이다. `effects io alloc` 과 `effects alloc io` 가 같은 해시를 낸다. 한때는 달랐다. 절의 원문을 해싱했기 때문이다. 그러면 내용 주소화가 뜻이 아니라 **철자**를 주소화한 것이 된다. 효과는 문법이 집합이라고 말하므로 정규화된 차례로 해싱한다.

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
```

실행 결과

```
$ lowentc --emit-db h4_contract.low
fn twice/1 iface:31247d9015a6c804 def:a088ec347ad280ef
```

계약도 표면이다. `requires le a 100 .` 을 붙이자 `iface` 가 바뀌었다. 부르는 쪽은 부르는 op 의 계약을 믿고 검사하며, 41장에서 보았듯 경계 검사를 지우는 근거가 계약이다. 계약이 바뀌었는데 해시가 같으면 캐시가 옛 계약을 믿은 결과를 그대로 쓰게 된다 — 전제가 바뀌었는데 증명을 재사용하는 셈이다. 한때 구현이 서명만 해싱해서 이 구멍이 있었고, 고쳤다.

49.4 증명할 수 있는 것과 없는 것

`LowentHash.v` 는 둘을 정확히 가른다. BLAKE3 의 충돌 저항성은 암호 가정이라 증명 대상이 아니다. 증명할 수 있는 것은 그 **앞**, 곧 정규 인코딩이 무엇을 지우고 무엇을 남기느냐다.

> **수학. 인코딩 정리(`LowentHash.v`)**
>
> `H1_comments_are_irrelevant` · `H2_body_does_not_touch_iface` · `H3_effects_are_a_set`, 그리고 `encoding_is_injective` — 인코딩이 같으면 인터페이스의 모든 조각이 같다. 마지막 정리가 값이다. "해시는 같은데 인터페이스가 다르다" 가 생긴다면 원인이 하나로 좁혀진다 — 해시 함수의 충돌. 인코딩이 애매해서 생긴 것이 아님을 기계가 보증한다.

계약 절은 효과와 달리 정규화하지 않고 **적힌 차례로** 해싱한다. 그래서 `requires A . requires B .` 와 `requires B . requires A .` 는 뜻이 같은데 해시가 다르다. 부정확하다. 그런데 안전한 방향이다. 같은 뜻이 다른 해시를 내면 재빌드가 한 번 더 돌 뿐이지만, 다른 뜻이 같은 해시를 내면 캐시가 거짓말을 한다. 이 방향이 안전하다는 것도 증명되어 있다. 구간 분석(41장)과 차용 검사(42장)에서 본 비대칭 — **모르면 넓게 잡는다** — 이 여기서도 판단을 정했다.

> **흔한 오해. 머리의 절 차례를 자유롭게 두면 해시가 흔들린다**
>
> 그래서 차례를 하나로 고정했다(3장). 다른 차례는 번역에서 거절되고 `--fmt` 가 고치므로, 같은 프로그램이 두 해시를 갖는 일은 없다. 한 뜻에 한 표기라는 원칙이 겉모습의 취향이 아니라 내용 주소화의 전제이기도 하다.

49.5 해시가 막는 것

상황	막는 것
주석을 고쳤는데 전체를 다시 짓는다	H1
본문을 고쳤는데 의존자의 의존자까지 다시 짓는다	H2
효과 차례만 바꿨는데 캐시가 빗나간다	H3
계약을 고쳤는데 의존자가 옛 결과를 쓴다	계약이 `iface` 에 들어 있다
의존이 조용히 바뀐다	해시가 내용을 담는 Merkle DAG
잠금 파일의 의존이 바뀐다	잠금 파일의 내용 해시가 거절한다(31장)

표 49.1 — 내용 주소화가 막는 것

뒤의 두 줄이 재현 가능한 빌드의 근거다. "같은 소스가 같은 결과를 낸다" 를 **비교할 수 있는 형태**로 만든 것이 내용 주소화의 값이다 — 믿는 것이 아니라 비교하는 것이 된다.

> **문. `--emit-db` 의 해시가 실제 빌드 결정에 쓰이는가?**
>
> **답.** 아직 완전히는 아니다. `--emit-db` 는 곁 파일이고, 이 판의 증분 빌드는 방출된 C 의 내용 해시로 돈다(31장). `iface·def` 해시는 무엇이 같고 다른지를 정확히 적어 두는 층이고, 시험이 그 같다·다르다의 관계를 지킨다. 해시가 빌드의 모든 결정을 맡는 것은 앞으로의 일이다.

49.6 증명하지 않은 것

- **정리는 모델 위의 것이다.** `LowentHash.v` 는 정규 인코딩에 대해 H1·H2·H3 을 증명했다. 실제 구현이 그 인코딩을 쓰는지는 시험이 몇 개의 예에서 확인한다.
- **정규화가 완전하지 않다.** 계약 절의 차례에 민감하고, 지역 변수 이름이나 수 리터럴의 표기(`0x10` 대 `16`)가 어디까지 정규화되는지 전수 확인하지 않았다.
- **BLAKE3 의 충돌 저항성은 가정이다.** 틀리면 캐시가 다른 내용을 같다고 본다.
- **덩어리(SCC) 해시가 여러 구현에서 같게 나온다는 것은 명세 수준의 약속**이고 확정되지 않았다. 순환을 다루는 방법은 여럿이다.
- **시험이 지키는 것은 해시의 값이 아니라 같다·다르다의 관계**다. 인코딩이 바뀌면 값은 바뀌지만 관계는 그대로여야 한다.

복습 정리

해시 함수의 충돌 저항은 가정이고, 해시가 의존의 해시를 담는 Merkle DAG 가 의존 추적을 대신하며, 순환하는 op 은 한 덩어리로 묶어 해싱한다. `--emit-db` 의 `iface·def` 해시는 주석에 반응하지 않고, 본문 변경은 `def` 만, 계약 변경은 `iface` 까지 바꾸며, 효과는 집합으로 정규화된다. 인코딩의 단사성은 증명되어 해시가 같은데 인터페이스가 다르면 원인은 충돌 하나로 좁혀진다. 계약 절을 정규화하지 않은 부정확함은 안전한 방향이다.

50 증명하지 않은 것

이 장의 차례

50.1 신뢰 기반 — 증명이 기대는 것 518
50.2 가장 큰 간극 — 모델과 구현 사이 519
50.3 남은 것 — 주제별로 520
50.4 “전면 정적 안전” 으로 팔지 않는다 521
50.5 그래서 무엇을 믿어도 되나 521

먼저 알아야 할 것

38장 왜 증명하는가 · 증명됨·전수 검사됨·스케치를 섞지 않는다
31장 짓고 시험하기 · 컴파일러를 맞대어 검증하는 다섯 방법
48장 문법과 해시의 증명 · 앞단은 가장 약한 이음매다

돌아보기

38장은 “증명된 언어로 짠 프로그램은 옳다” 가 왜 오개념이라고 했는가?

> **답.** 증명은 **언어의 규칙**에 대한 것이지 **당신의 프로그램**에 대한 것이 아니기 때문이다. 차용 규칙이 건전하다는 정리는 번역을 통과한 프로그램에 차용 위반이 없다는 것만 말한다. 이 장은 그 경계를 끝까지 따라가, 증명이 기대는 것과 덮지 않는 것을 한곳에 모은다.

이 장의 필요성과 맥락

증명을 자랑하는 문서는 대개 두 가지 가운데 하나를 한다 — 경계를 적지 않거나, 맨 앞에 한 줄로 면피하고 잊는다. 둘 다 읽는 사람을 잘못된 확신으로 데려간다. **증명의 값은 그것이 무엇을 덮지 않는지 아는 데서 나온다.** 이 장이 책의 마지막인 것은 편집상의 우연이 아니다. 앞의 장들이 무엇을 주장했는지 알고 나서야 그 경계가 뜻을 갖는다. 이 장을 읽지 않고 이 책을 인용하면, 이 책이 말하지 않은 것을 말한 것으로 만들게 된다.

이 장이 끝나면

증명이 기대는 신뢰 기반(Coq 커널·C 컴파일러·BLAKE3·빌린 증명 따위)과 그것이 틀렸을 때 무너지는 범위를 알게 된다. 모델과 구현 사이의 간극을 잇는 장치들이 경험적인지 정적인지 가르고, 가장 약한 이음매가 앞단이라는 것을 이해한다. 주제별로 남은 것, 이 언어를 “전면 정적 안전” 이라 부르면 안 되는 이유, 그리고 결국 무엇을 믿어도 되는지를 정리한다.

이 장에서 답할 질문

1 그렇다면 이 모든 증명은 결국 무엇을 사는가?

50.1 신뢰 기반 — 증명이 기대는 것

믿는 것	왜 믿나	틀리면

Coq·Rocq 의 커널	작은 핵이고, 두 판(8.20·9.2)이 같은 답을 낸다	모든 정리가 무의미해진다
Iris	락 증명에만 필요하다	락 정리만 무너진다
iRC11 · gpfsl · Iris 의 `rw_spin_lock`	약한 메모리·SPSC·rwlock 읽기 쪽의 증명을 빌렸다	그 정리들만 무너진다
C 컴파일러(gcc·clang)	대안이 없다	네이티브 코드가 중간 표현과 달라진다
BLAKE3 의 충돌 저항	내용 주소화의 전제다	캐시가 다른 내용을 같다고 본다
운영체제·하드웨어	대안이 없다	전부
`proven_c_lib`(기반 라이브러리)	자체 시험이 있다	자료구조가 조용히 틀린다
모델이 언어를 옳게 담았다	사람이 읽어서 확인한다	아래의 간극

표 50.1 — 신뢰 기반

50.2 가장 큰 간극 — 모델과 구현 사이

```
정리는 모델에 대한 것이다.
모델은 사람이 쓴 것이다.
컴파일러는 다른 언어로 쓴 다른 코드다.
그 둘이 같다는 것은 증명되지 않았다.
```

이음매를 잇는 장치들이다.

장치	무엇을 하나	성격
유계 전수 모델 검사	작은 크기의 모든 경우에서 모델의 판정과 구현의 판정이 같은지	경험적
실제 프로그램에서 뽑은 사건	진짜 프로그램의 차용 사건을 같은 모델에 먹인다	경험적 · 실제 프로그램
두 백엔드 대조	VM 과 네이티브가 같은 답을 내는지	경험적
분석 자기 고발(`E-VM-ANALYSIS`)	컴파일러의 “지워도 된다” 를 실행으로 반박한다	경험적
증명서 검산	지운 검사의 근거를 독립 검산기가 산술로 다시 따진다	정적
회귀 시험	출력이 바뀌면 보인다	회귀만

표 50.2 — 모델과 구현을 잇는 장치

앞의 넷은 경험적이다. “지금 출력이 옳다” 가 아니라 “이 입력들에서 어긋나지 않았다” 를 말한다. 증명서 검산은 성격이 다르다. 그 자리를 때리는 입력이 없어도 말한다. 다만 그것도 **사실을 믿고 추론만** 검사한다. 두 층은 서로 다른 것을 잡는다. 증명이 못 잡은 결함을 대조가 잡았고, 대조가 못 잡은 결함을 증명이 잡았다. 겹쳐 쓰는 이유다.

가장 약한 이음매는 앞단(어휘·파서)이다. 두 백엔드 대조는 VM 과 네이티브를 비교하지만 둘은 같은 앞단을 지난다. 앞단이 틀리면 둘 다 똑같이 틀린다. 파서를 문법 모델과 직접 대조하려던 시도는 모델이 실제 문법보다 작아서 결론을 낼 만큼의 짝을 얻지 못했다. 이 자리를 닫으려면 앞단을 독립적으로 한 벌 더 구현해 대조해야 하고, 그 비용(두 벌을 유지하는 것)은 아직 치르지 않았다.

실제 사례. 이 책이 찾은 어긋남들

이 책의 예제는 모두 VM 과 네이티브로 돌려 맞대고, 거절 예제는 진단 코드까지 확인했다. 그 과정에서 컴파일러와 명세의 어긋남이 스무 건 가까이 드러났다. 몇 가지는 본문에 적었다 — `method` 로 부른 op 의 효과가 부르는 쪽에 번지지 않는 것(23장), 식이 든 `ensures` 가 경고 없이 검사되지 않는 것(14장), `arg` 의 번호가 두 백엔드에서 다른 것(2장), `expr` 섬에서 비교와 `and` 의 우선순위가 명세표와 다른 것(8장). 모두 증명의 **모델**이 아니라 **구현**과 **문서** 쪽의 어긋남이다. 예제를 실제로 돌려 싣는다는 규율이 없었다면 이 책은 그 어긋남을 사실처럼 적었을 것이다.

50.3 남은 것 — 주제별로

무엇	상태
효과 체계	증명됨. `task_group` 이 `concurrent` 를 흡수하는 규칙과 효과 닫힘은 모델 밖
효과 기반 최적화	증명됨(재정렬·공통 부분식 제거·기억·죽은 코드 제거). 호출 경계와 멈추는 시점은 밖
반복의 종료	다루지 않는다. 무한 반복은 정당한 프로그램이다
타입 체계 전체의 건전성	부분. effect-row 는 없고 증명도 없다
정규화의 완전성	부분. 효과는 집합으로 정규화(증명됨). 계약 절은 적힌 차례에 민감하고 그것이 안전한 방향임을 증명했다
덩어리 해시의 다구현 호환	미확정. 명세 수준의 약속
파서 ≡ 문법 모델	미증명

표 50.3 — 언어 코어와 도구

무엇	상태
RC11 일반 약한 차례	부분. 모델 안의 메타정리는 증명됨. 특정 락 없는 알고리즘을 약한 차례로 검증한 것은 SPSC 하나(빌린 증명)
교착 자유	증명됨. 단 잠금 오름차순 규율은 사람이 지켜야 한다(도구가 강제하지 않는다)
굶주림 없음	증명됨. 라운드로빈·양보를 가정하고, 우선순위와 차단은 밖
level 1 병렬의 구현	구현됨. 정리는 구현을 검증하지 않는다
락 없는 자료구조	SPSC 링 버퍼 하나. MPSC·MPMC·seqlock 은 빌릴 증명이 없어 넣지 않았다
원자 차례 → 기계어	C11 atomic 으로 1:1 방출하므로 나머지는 C 컴파일러(신뢰 기반)의 몫. 매핑 자체는 시험으로 확인

표 50.4 — 동시성

무엇	상태
부동소수의 성질(반올림·NaN·−0)	다루지 않았다
구간 분석의 부동소수 지원	없다
`page_fault`·`blocking` 따위 효과 원자	어휘만 있고 원시 연산이 없다
effect-row 다형성	없다

표 50.5 — 기능 범위

```
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
```

실행 결과

```
$ lowentc --run tenths floats.low
tenths() = 0
$ lowentc --run close_enough floats.low
close_enough() = 1
```

0.1 과 0.2 를 더한 값은 0.3 과 같지 않다. 이것은 결함이 아니라 IEEE 754 의 성질이고, 그 성질은 이 언어의 증명이 다루지 않았다. 부동소수의 같음은 `close_enough` 처럼 허용 오차로 묻는다. 부동소수가 걸린 계산에서 이 책의 "증명됨" 은 아무것도 보증하지 않는다.

50.4 "전면 정적 안전" 으로 팔지 않는다

```
정적으로 막는 것   차용 · 영역 · 참조 탈출 · 병렬 겹침 · 정수 넓히기
실행이 막는 것     증명 안 된 자리의 경계 검사 · 넘침 · 계약 검사
                   세대 핸들의 늘어진 참조
```

이 언어는 **섞어 쓰는** 설계다. 그리고 그것을 정적이라고 부르지 않는다. 비교를 정직하게 적으면, Rust 는 RustBelt 로 증명이 있고, Zig 는 안전을 주장하지 않는다(그것도 정직한 태도다). Lowent 는 "증명 있음 — **주로 순차 단편에 대해**" 다. 그 조건절을 떼면 거짓말이 된다. 그리고 `release_fast` 빌드 모드는 남은 계약 검사를 없애 신뢰 경계를 하나 더 연다(14장). "미정의 동작이 없다" 는 말은 안전한 부분집합 안에서만 참이다.

> **문. 그렇다면 이 모든 증명은 결국 무엇을 사는가?**
>
> **답.** 두 가지를 산다. 첫째, 컴파일러가 **무엇을 거절해야 하는지**가 감이 아니라 논증으로 정해진다. 규칙이 하나로 충분한지, 조건이 ≤ 인지 < 인지가 증명에서 나온다. 둘째, 구현이 겪은 결함이 **정리**로 남는다. 이야기는 잊히지만, 누가 그 검사를 지우면 증명이 깨진다. 증명은 프로그램이 옳다고 보증하지 않지만, 언어가 조용히 틀리는 자리를 체계적으로 줄인다.

50.5 그래서 무엇을 믿어도 되나

```
✔ 정수가 조용히 값을 바꾸는 일은 없다              (수 · 증명됨)
✔ 번역을 통과한 코드에 차용 위반은 없다             (소유 · 증명됨, 순차)
✔ 효과 선언이 실제로 일어나는 효과를 덮는다         (효과 · 증명됨, 모델 안)
✔ 안전 코드에 데이터 경합은 없다                   (동시성 · 증명됨)
✔ 병렬 결과는 순차와 비트까지 같다                 (병렬 · 증명됨)
```

```
✔ 기본 차례면 순서대로 사고해도 된다              (약한 메모리 · 증명됨, ∀실행)
✔ 경계 검사를 지운 자리는 범위 안임이 논증된 자리  (경계 · 증명됨 + 검산)

△ 컴파일러가 이 모델을 정확히 따라간다            (경험적)
△ 약한 차례를 명시한 코드                        (SPSC 하나만 증명 – 나머지는 감사 대상)
✘ 부동소수의 세부                                 (다루지 않았다)
```

마지막 줄들이 있는 문서를 믿어도 된다. 없는 문서는 다시 봐야 한다.

흔한 오해. ✔ 가 붙은 줄은 이 판의 컴파일러가 언제나 그렇게 한다는 뜻이다

✓ 는 **모델**에 대한 정리이고, 컴파일러가 그 모델을 따라가는지는 △ 줄 — 경험적 — 에 속한다. 이 매뉴얼을 쓰며 그 틈을 여럿 재었다. `reduce` 의 누산 시작값이 항등원이 아니면 나누어 돈 네이티브 답이 순차와 다르고(27장), `let` 이름을 `mut_ref` 로 빌려주면 불변이 뚫리며 (12장), 권한 자리에 수를 넘기면 권한 없는 op 이 출력한다(16장). 모두 그 장에 경고로 적었고 개발 저장소에 결함으로 올렸다. ✓ 는 "이것이 막혀야 한다" 는 근거이고, 막혔는지는 그 장의 경고와 결함 목록을 함께 본다.

복습 정리

증명은 Coq 커널·C 컴파일러·BLAKE3·빌린 증명·운영체제 같은 신뢰 기반에 기댄다. 가장 큰 간극은 모델과 구현 사이이고, 전수 검사·대조·자기 고발은 경험적으로, 증명서 검산은 정적으로 그 사이를 잇는다. 가장 약한 이음매는 앞단이다. 반복의 종료·부동소수·effect-row·파서의 모델 대응은 증명 밖이다. 이 언어는 정적 검사와 실행 검사를 섞은 설계이고, 증명은 주로 순차 단편에 대한 것이다. 믿어도 되는 것과 경험적인 것과 다루지 않은 것을 갈라서 읽는다.

부록

부록 A — 낱말과 내장 연산

낱말을 **설명**하는 곳은 본문이다. 이 부록은 코드를 읽다 막혔을 때 훑어볼 표만 모은다. 낱말 목록은 명세의 부록 A 와 같고, 그 목록에 없는 것은 낱말이 아니다.

낱말 마흔셋

갈래	낱말	다루는 장
선언	`module use type newtype struct enum trait contract actor state`	21·13장
op	`fn proc export unsafe extern satisfies`	5장
지역	`let var set be`	6장
흐름	`if else while for guard match case return break continue try`	7장
식과 값	`expr make true false none`	8장
동시성	`spawn send`	25장
소유	`drop`	19장
시험	`test expect`	31장
블록	`do end`	3장

표 50.1 — 낱말의 전부

`rem` 과 `note` 는 주석을 여는 표시이고, `input·output·effects·requires` 같은 절 머리와 `add·len` 같은 내장 연산은 낱말이 아니지만 이름으로 쓸 수 없다.

없앤 낱말

없앤 것	대신
`loop`	`while true .`
`give`	`return`
`unit`	`void`
`calcop · procop`	`fn · proc`
`is · as`(선언 속) `· local`	적지 않는다
`to · in`(칸 접근)	`field a b · index a i`
`on`	액터 안의 `proc`
`fail`	`return error <갈래>`
`; · ,`	`.`

표 50.2 — 없앤 낱말과 대신 쓰는 것(`E-VOCAB-REMOVED`)

`as` 와 `to` 는 `use … as <별칭>` 과 `case <아래> to <위>` 의 자리 표식으로만 남아 있다.

자주 쓰는 내장 연산

갈래	연산

산술(멈춤)	add sub mul div mod neg abs min max
산술(처분을 고름)	wrap_* sat_* chk_* div_nz nonzero_of
폭 바꾸기	widen narrow narrow_wrap narrow_sat narrow_try cast bit_cast
비교와 논리	eq ne lt le gt ge and or not
비트	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
줄	len index subslice view view_array same_slice
묶음	field method isa get
답을 담는 타입	some ok error is_some is_none is_ok is_error some_value ok_value error_value value_or
번역 시점	size_of comptime config
할당	alloc_bytes
입출력 잎	write_out read_in arg · 파일·연결·시계·난수·터미널 잎은 표준 라이브러리가 감싼다
원자	atomic_load atomic_store atomic_add atomic_sub atomic_swap atomic_cas atomic_fence
장치	read_volatile write_volatile
동시성	await drain channel chsend chrecv yield spawn send
원자(비트)	atomic_and atomic_or atomic_xor
부동소수 수학 — 4장	sqrt sin cos exp log pow floor ceil round fmod sum_neumaier sum_seq
파이프 단계 — 24장	pipe map filter fold scan take skip zip enumerate reverse collect count all any into
레인(SIMD) — 27장	splat load store load_masked store_masked select reduce_add reduce_mul reduce_min reduce_max avg native_lanes rotate prefetch
작은 수의 집합 — 13장	bitset_new union intersect difference complement contains is_subset is_empty remove
배치와 뷰 — 13장	encode try_view view_segments seg segs capacity
빌림과 영역 — 12장 · 18장	ref mut_ref deref borrow region stack_new push pop swap
계약과 오류 — 14장	range ret expect panic
계산 잎 — call_builtin 뒤에서만 선다	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 문자열 — 29장	cstr_of str_from_cstr
호스트 잎 — 32장	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

표 50.3 — 내장 연산의 갈래

★ **계산 잎 열다섯은 전역 어휘가 아니다.** `call_builtin sha256 msg out` 처럼 그 자리에서만 선다 — 맨몸으로 부르면 `E-BUILTIN-BARE`, 모르는 이름을 대면 `E-BUILTIN-NAME` 이다. 까닭은 하나다: 프로그램이 한 번 쓰

는 낱말로 전역 어휘가 늘면 읽는 사람이 외울 것이 는다. 같은 규율이 `pipe` 안의 단계 이름과 `cast u8 x` 의 타입 자리에도 이미 있다.

이 표는 정본의 내장 연산을 빠짐없이 무리로 나눈 것이다. 무리 이름 옆의 장이 그 무리를 예제와 함께 설명한다. 한 줄씩의 뜻은 저장소의 `docs/spec/BUILTIN-MEANINGS.tsv` 가 정본이다.

효과와 권한

효과	허락하는 권한	다루는 장
`io`	`cap io` · `file_system` · `net` · `tty` · `clock` · `random`	16장
`alloc` · `heap`	`cap allocator` · `cap heap`	18·20장
`atomic`	`cap atomic`	27장
`device`	`cap mmio`	30장
`unsafe`	`cap c` · `cap machine`(쓰는 자리에 따라)	29·30장
`state` · `panic` · `wait` · `concurrent`	권한 없이 적는다	15·26장

표 50.4 — 효과 원자와 짝이 되는 권한

부록 B — 진단 찾아보기

앞의 두 표는 이 책의 예제가 실제로 일으킨 진단이다. 예제에는 안 나오지만 만날 수 있는 진단은 맨 끝 «그 밖의 진단» 에 모았다. 코드는 판이 바뀌어도 같은 뜻을 지킨다. 진단의 긴 영문 설명과 고치는 법은 컴파일러가 그 자리에서 말한다(`--diag-json` 은 기계가 읽는 꼴로).

번역할 때

코드	뜻	보이는 곳
E-ACCESS-MODE	access 로 적은 읽기·쓰기 약속을 몸이 어긴다	27장
E-ACTOR-FIELD	액터의 상태 칸을 바깥에서 읽는다	25장
E-ACTOR-STATE-REF	액터 상태 칸에 빌림을 둔다	25장
E-ACTOR-UNINIT	갓 띄운 액터의 슬라이스 칸을 세우기 전에 읽는다	20장
E-ALLOC-AMBIGUOUS	맞는 할당기가 둘 이상인데 using 으로 고르지 않았다	20장
E-ALLOC-NESTED	같은 뿌리의 안쪽 영역이 열린 채 바깥 출처로 깎았다	18장
E-ALLOC-NOCAP	alloc 을 적었는데 할당 권한을 받지 않았다	16장
E-ALLOC-NOSOURCE	할당기를 쓰는 호출인데 이 op 안에 맞는 할당기가 하나도 없다	20장
E-ALLOC-OUTLIVES	영역의 바이트를 영역 밖에서 태어난 액터에게 건넨다	18장
E-ALLOC-SHARED	원자적이지 않은 할당기를 태스크에 건넨다	26장
E-ALLOC-TASK	뿌리에서 깎는 op 을 태스크로 띄운다	26장
E-ALLOC-USING-UNUSED	할당기를 쓰지 않는 호출에 using 을 적었다	20장, 34장
E-ASM-TARGET-UNKNOWN	asm 절의 기계 이름이 대상 목록에 없다	30장
E-ASM-UNBOUND	어셈블리 템플릿이 선언하지 않은 피연산자를 부른다	30장
E-ATOMIC-NOCAP	atomic 을 적었는데 cap atomic 을 받지 않았다	27장
E-ATOMIC-ORDER	그 원자 연산에 뜻이 없는 기억 차례다	27장
E-BLOCK-UNCLOSED	do 로 연 블록을 end 로 닫지 않았다	2장
E-BOUND-UNSAT	타입 인자가 요구한 트레이트를 갖추지 못했다	22장, 23장
E-BRAND-REUSED	같은 브랜드로 저장소를 두 번 열었다	35장
E-CAP-FORGE	권한 칸을 가진 액터를 그 권한 없이 띄운다	16장, 20장, 25장
E-CAP-KIND	다른 종류의 권한을 건넸다	16장
E-CAP-LOCAL	받은 권한을 지역 이름에 옮겨 담았다	16장
E-CAP-MISSING	권한이 필요한 내장 연산에 권한을 첫 피연산자로 적지 않았다	16장

E-CHAR	이 언어에 없는 글자(기호)다	6장, 7장, 8장, 9장
E-CHAR-WIDTH	글자 리터럴이 글자 하나가 아니다	3장
E-CLAUSE-ORDER	op 머리의 절이 정해진 차례를 어겼다	2장, 3장, 22장
E-COMPTIME-ARG	comptime 자리에 실행 값을 주었다	22장
E-CONC-ALONE	묶음이 흐름 하나만 만들고 그 흐름이 짝을 기다린다	26장
E-CONC-DEADLOCK	묶인 흐름이 모두 받기만 하고 아무도 보내지 않는다	26장
E-CONFIG-TYPE	구성이 손잡이의 선택지에 없는 값을 준다	31장
E-CONFIG-UNDEF	선언하지 않은 빌드 손잡이를 구성이나 config 가 부른다	31장
E-CONTRACT-DEAD	requires 가 이미 배제한 오류를 선언했다	14장
E-CONTRACT-IMPOSSIBLE	상수끼리의 호출이 상대의 requires 를 어긴다	14장, 35장
E-CONTRACT-UNSAT	함께 참일 수 없는 전제 둘	14장
E-DEP-MISSING	use … from 의 자리에서 파일을 읽을 수 없다	21장
E-EFFECT	선언하지 않은 효과를 낸다	15장, 20장, 26장
E-EFFECT-CALC	순수한 fn 이 효과를 낸다	1장, 5장, 15장, 16장, 18장, 24장
E-EFFECT-DUP	효과 줄에 같은 원자를 두 번 적었다	15장, 44장
E-EFFECT-NO-CAP	io 따위를 적었는데 허락하는 권한이 없다	2장, 16장
E-EFFECT-NONE-MIX	none 을 다른 효과와 함께 적었다	15장
E-EFFECT-PURITY	fn 이 호출자나 액터가 볼 수 있는 상태에 쓴다	5장, 15장, 25장
E-EFFECT-REDUNDANT	fn 에 effects none 을 적었다	5장
E-EFFECT-UNDEF	목록에 없는 효과 낱말이다	15장, 44장
E-ENS-UNDEF	ensures 가 없는 이름을 가리킨다 — 돌려주는 값은 ret	14장
E-ENTRY-PARAMS	시작점이 권한 아닌 입력을 받는다	16장
E-ENUM-DOT	열거의 갈래를 마침표로 닫지 않았다	10장
E-ENUM-INFINITE	갈래가 자기 타입을 값으로 품는다	10장
E-ERR-UNDECLARED	errors 절에 없는 오류를 돌려준다	11장
E-ERRORS-STATE	errors 조건이 액터의 상태 칸을 읽는다	25장
E-ESCAPE	지역을 가리키는 참조가 op 밖으로 나간다	12장
E-EXCL	같은 값에 대한 빌림이 겹친다	12장, 20장, 26장, 43장
E-EXPR-APP	expr 섬 안의 부름을 괄호로 묶지 않았다	8장
E-EXPR-CHAIN	expr 섬에서 비교를 이어 썼다	3장
E-EXPR-UNARY	expr 섬에 단항 연산자를 썼다	8장
E-FFI-LINK	extern op 이 부를 씨 심볼을 적지 않았다	29장
E-FFI-NOCAP	C 를 부르는데 cap c 를 받지 않았다	29장
E-FFI-NOEFFECT	C 를 부르는 op 에 효과 줄이 없다	29장
E-FFI-NOUNSAFE	C 를 부르는데 unsafe 표시가 없다	29장

E-FFI-TYPE	C ABI 가 표현할 수 없는 타입이 경계를 건넌다	29장
E-FIELD-GLUED	값 뒤에 점을 붙여 칸을 읽으려 했다 — `field` 로 적는다	3장
E-FN-CAP	되부름으로 넘길 op 이 권한을 요구한다	29장
E-FN-NOTEXPORT	`unsafe_fn` 이 `export extern` 이 아닌 op 을 가리킨다	29장
E-FOLD-OP	`pipe` 스테이지가 없는 op 을 부른다	24장
E-FOLD-ORDER	fold 단계 op 의 누산값 자리가 결과 타입과 다르다	24장
E-GROUP-UNCLOSED	괄호가 닫히지 않았다	3장
E-GUARD-FALLTHROUGH	`guard` 의 `else` 가 떠나지 않는다	7장
E-HEAP-NOHOST	운영체제 없는 대상에서 자라는 뿌리를 청한다	18장
E-IF-VALUE	`if` 를 값으로 썼다	6장
E-IMMUTABLE	`let` 으로 지은 이름에 `set` 했다	6장
E-IR-ARITY	op 을 인자 수가 맞지 않게 불렀다	5장, 48장
E-IR-UNDEF	그 자리에서 알 수 없는 이름이다	6장, 7장, 21장, 22장, 25장
E-ISR-CALLED	인터럽트 처리기를 코드에서 불렀다	30장
E-ISR-EFFECT	인터럽트 처리기가 `effects device` 를 적지 않았다	30장
E-ISR-PARAMS	인터럽트 처리기가 매개변수를 받는다	30장
E-LET-NOVALUE	`be` 뒤에 값이 없다	6장
E-LOCK-NOTYET	흐름끼리 나누는 자물쇠 타입은 아직 짓지 않았다	26장
E-MATCH-INEXHAUSTIVE	`match` 가 모든 경우를 덮지 않는다	7장, 10장
E-MATCH-REDUNDANT	`match` 의 갈래가 영영 돌지 않는다(_ 뒤의 갈래·겹친 범위)	11장
E-METHOD-RECV	붙은 op 의 수신자가 첫 입력이 아니다	23장
E-METHOD-UNDEF	수신자의 타입에 그 이름의 붙은 op 이 없다	22장, 23장
E-MMIO-BYVALUE	레지스터 묶음을 값으로 받는다	30장
E-MMIO-PERM	읽기 전용 레지스터에 쓴다	30장
E-MONO-NOTYPE	제네릭 부름에 앞자리 타입 인자가 없다	22장
E-MREF-SLICE	`mut ref slice` 를 썼다	12장
E-NAME-BUILTIN	내장 op 의 이름을 선언이나 지역 이름으로 썼다	5장, 6장
E-NAME-DUP	한 모듈에 같은 이름을 두 번 선언했다	21장
E-NAME-SCOPE	블록 안에서 지은 이름을 블록 밖에서 읽는다	6장
E-NAME-SHADOW	살아 있는 이름을 다시 지었다(가림)	3장, 6장

E-OPT-UNUSED	선언한 빌드 손잡이를 아무 코드도 읽지 않는다	31장
E-OWN-INCOMPLETE	완결이 필요한 값을 끝내지 않고 버린다	19장, 28장, 36장
E-OWN-JOIN	갈래마다 소유 상태가 다르다	19장
E-OWN-MOVED	옮긴 값을 다시 쓴다	19장, 25장, 28장, 35장, 36장
E-PAR-ASSOC	모으는 연산이 결합적이지 않다	27장
E-PAR-CARRY	나누는 되풀이가 걸음을 넘어 사는 자리에 쓴다	27장
E-PAR-FLOAT	부동소수 누적을 나누어 모으려 했다	27장
E-PAR-IDENTITY	`reduce` 의 시작값이 그 연산의 항등원이 아니다	27장
E-PAR-NOLOOP	`parallel` 절이 나눌 되풀이를 찾지 못했다	27장
E-PAR-READ	나누는 되풀이가 남의 원소를 읽는다	27장
E-PAR-WRITE	나누는 되풀이가 남의 원소에 쓴다	27장, 45장
E-PIPE-NO-TERMINAL	종결자 뒤에 스테이지가 온다	24장
E-PIPE-STAGE	`pipe` 스테이지 목록에 없는 낱말이다	24장
E-PROFILE-LEVEL	빌드 프로파일이 주지 않는 동시성을 쓴다	25장
E-REGION-ESCAPE	영역에서 얻은 바이트를 밖으로 들고 나간다	18장
E-RETURN-PARTIAL	어떤 길에서 값을 돌려주지 않는다	3장, 5장, 7장
E-SPAWN-SCOPE	`task_group` 밖에서 흐름을 만든다	26장
E-STMT-ELSE	`else` 가 블록 안에 있다	7장
E-TIER-EFFECT	선언한 기계 등급이 감당하지 못하는 효과다	30장
E-TOPLEVEL	최상위에 올 수 없는 것이 최상위에 있다	3장
E-TRAIT-EFFECT	갖춘 op 이 서명보다 많은 효과를 가진다	23장
E-TRAIT-MISSING	트레이트의 op 하나가 없다	23장
E-TRAIT-SIG	서명에 `fn`·`proc` 을 적었거나 매개변수 수가 다르다	23장
E-TRY-NORESULT	`try` 를 썼는데 이 op 은 그 오류를 돌려줄 수 없다	11장
E-TYPE-ARG	인자의 타입이 매개변수와 맞지 않는다	12장
E-TYPE-ARGMUT	읽기만 되는 값을 고치는 자리에 넘긴다(`let` 을 `mut_ref` 로)	9장, 12장
E-TYPE-ARRAY	`array` 의 길이를 타입 뒤에 적었다	9장
E-TYPE-BITCAST	`bit_cast` 의 목표가 모든 비트열이 값인 타입이 아니다	20장
E-TYPE-COLLECT	`collect into` 가 넓은 값을 좁은 그릇에 담는다	24장
E-TYPE-COND	조건 자리에 참거짓이 아닌 값이 왔다	4장
E-TYPE-DECL	타입 선언에 `be` 를 끼웠다	13장
E-TYPE-FIELD	구조체를 만들 때 칸이 빠졌거나 없는 칸을 적었다	10장
E-TYPE-INSTANCE	같은 제네릭의 서로 다른 인스턴스를 섞었다	35장

E-TYPE-KIND	그 연산에 쓸 수 없는 갈래의 타입이다(예: `bool` 을 `cast`)	8장, 10장, 13장
E-TYPE-MIX	부동소수와 정수 리터럴을 섞어 셈한다	4장
E-TYPE-MUT	`mut` 이 아닌 슬라이스의 원소에 쓴다	9장
E-TYPE-NOMINAL	표현이 같아도 이름이 다른 타입을 섞었다	13장
E-TYPE-REF	읽기 참조로 쓴다	12장
E-TYPE-REFVAL	참조를 값처럼 썼다	12장
E-TYPE-RETURN	돌려주는 값이 op 의 출력 타입과 다르다	11장, 31장
E-TYPE-SIGN	값을 지키는 넓히기가 없는 부호 섞기다	4장, 39장, 40장
E-TYPE-WIDTH	값이 선언된 타입에 들어가지 않는다	3장, 4장, 13장
E-UNSAFE-UNDECLARED	`unsafe` 효과를 적었는데 op 에 `unsafe` 표시가 없다	29장
E-VEC-SPLAT	splat 을 식 안에 바로 적었다	27장
E-VISIBILITY	다른 모듈의 감춘 이름에 닿는다	21장
E-VOCAB-REMOVED	없앤 낱말이나 철자다	3장, 5장, 7장, 15장, 48장
E-WIDEN-SIGN	부호 있는 수를 부호 없는 타입으로 `widen` 했다	13장
W-EFFECT-OVER	선언만 하고 내지 않는 효과다(경고)	15장, 16장
W-EXPORT-HIDDEN	내보낸 서명이 감춘 타입을 쓴다	21장
W-NOT-YET	이름은 받지만 아직 뜻이 없는 낱말을 썼다	13장
W-USE-EXTERNAL	번역 단위에 없는 모듈을 들여온다(경고)	32장

표 50.1 — 예제가 보이는 진단

실행 중에 멈출 때

코드	뜻	보이는 곳
E-VM-OVERFLOW	정수가 선언된 폭에서 넘쳤다	4장
E-VM-DIV0	0 으로 나누었다	4장
E-VM-CAST	좁히기·`cast` 의 값이 목표 타입에 들어가지 않는다	4장
E-VM-SHIFT	옮기는 칸 수가 폭 이상이다	4장
E-VM-BOUNDS	색인이 범위 밖이다	9장
E-VM-NONE	없는 값을 꺼냈다	11장
E-VM-CONTRACT	계약(`requires`·`ensures`·`errors`·범위)이 깨졌다	14장
E-VM-PANIC	`panic` 을 불렀다	7장
E-VM-ANALYSIS	지운 검사의 자리가 실제로 범위 밖이다(컴파일러 결함)	40장
E-TEST-FAIL	시험의 `expect` 가 거짓이다	31장

표 50.2 — 실행 중의 진단

그 밖의 진단

예제에는 나오지 않지만 쓰다 보면 만날 수 있는 진단이다. 대부분은 `--run` 으로 돌릴 때 **VM(실행기)** 이 멈추며 낸다. VM 은 시험과 대조를 위한 실행기라 자원을 고정된 크기로 잡아 두고, 그 한도를 넘으면 **조용히 틀린 값을 내는 대신 멈춘다**. 한도는 프로그램의 결함이 아니라 VM 의 크기이므로, 같은 프로그램을 네이티브로 지으면 대개 문제없이 돈다.

코드	뜻과 고치는 법
`E-IR-EXTRA`	`expr` 섬 뒤에 남는 피연산자가 있다 — 섬은 식 하나로 끝난다. 괄호로 묶거나 줄을 나눈다
`E-IR-LIMIT`	한 번역 단위의 op 이 도구가 나를 수 있는 수보다 많다 — 모듈을 나눈다
`E-IR-LOCALS`	한 op 의 지역 이름이 너무 많다 — op 을 쪼갠다
`E-MONO-FIXPOINT`	제네릭 단형화가 끝나지 않는다(타입이 계속 자란다) — 스스로를 더 큰 타입으로 부르는 제네릭을 끊는다
`W-COL0`	맨 위 선언이 0 열에서 시작하지 않는다 — 들여쓰기가 틀려 앞 선언의 몸에 들어간 줄일 수 있다
`W-UNBOUND`	이름 붙인 `contract` 를 선언했는데 어느 op 도 `satisfies` 로 쓰지 않는다
`W-RFC-PENDING`	설계가 아직 RFC 단계인 기능을 썼다 — 프로그램에 고칠 것은 없다
`W-CBE-SLOW`	이 op 이 네이티브에서 느린 길(태그 값)로 낮춰진다 — 까닭은 `--why-slow` 가 op 마다 말한다

표 50.3 — 번역할 때 — 예제에 없는 것

코드	뜻과 고치는 법
`E-VM-ERR`	오류인 `result` 에서 `ok_value` 로 값을 꺼냈다 — 먼저 `is_ok` 로 묻거나 `try` 로 넘긴다
`E-VM-DANGLING`	빌린 참조가 그 자리의 수명보다 오래 쓰였다(돌려준 뒤에 씀)
`E-VM-EXCL`	한 자리를 쓰는 빌림과 다른 빌림이 겹쳐 쓰였다 — 먼저 빌린 것을 끝내고 다음을 빌린다
`E-VM-READONLY`	읽기 전용 참조(`ref`)로 썼다 — 쓰려면 `mut_ref` 로 빌린다
`E-VM-FIELD`	레코드 · 레지스터 블록에 없는 칸을 읽었다
`E-VM-AWAIT`	`await` 에 작업 핸들이 아닌 값을 주었다 — 핸들은 `spawn <op>` 이 준다
`E-VM-CHAN-EMPTY` · `E-VM-CHAN-FULL`	빈 채널에서 받거나 찬 채널에 보냈는데, 기다려 줄 스케줄러가 없다 — 영영 풀리지 않는 대기다
`E-VM-SCHED-LOOP`	핸들러가 서로 메시지를 끝없이 주고받아 스케줄러가 멈추지 못한다
`E-VM-FD` · `E-VM-FMODE` · `E-VM-FWHENCE`	쓸 수 없는 파일 번호(1 · 2 만 쓴다) · 없는 열기 방식(0 · 1 · 2) · 없는 `seek` 기준(0 · 1 · 2)
`E-VM-FHANDLE` · `E-VM-SHANDLE` · `E-VM-REACTOR`	열린 파일 · 소켓 · 리액터 핸들이 아닌 값을 넘겼다
`E-VM-EXTERN`	몸이 C 에 있는 `extern` op 을 VM 에서 불렀다 — 네이티브로 지어서 돌린다

E-VM-MMIO · E-VM-RESERVE	절대 주소의 레지스터 블록 · 링커가 주는 저장소를 VM 에서 열었다 — 실제 기계에서만 뜻이 있다

표 50.4 — 실행 중에 — 프로그램의 잘못

코드	무엇이 다 찼나
E-VM-STACK · E-VM-DEPTH · E-VM-OOM	피연산자 스택 · 호출 깊이 · 호출 스택을 키울 메모리 — 재귀가 너무 깊다
E-VM-BOXPOOL · E-VM-RECPOOL · E-VM-STKPOOL · E-VM-BSETPOOL	some·ok 로 감싼 값 · 구조체 값 · 스택 컨테이너 · 비트셋을 담는 풀 — 한꺼번에 살아 있는 값을 줄인다
E-VM-CHAN · E-VM-MBOX	채널 64 개 · 비동기 우편함 4096 칸 — 다시 쓰지 않으므로 한 실행에서 그만큼만 만든다
E-VM-FILEPOOL · E-VM-SOCKPOOL	동시에 열 수 있는 파일 · 소켓 수 — 다 쓴 것을 닫는다
E-VM-CHAIN-LIMIT	VM 안에서 VM 을 부르는 겹침이 너무 깊다
E-VM-BUDGET	대조(오라클) 실행의 걸음 예산을 다 썼다 — 평소 --run 에서는 나지 않는다

표 50.5 — 실행 중에 — VM 의 한도(프로그램의 결함이 아니다)

부록 C — 흔한 실수와 고치는 법

다른 언어의 버릇으로 Lowent 를 쓸 때 자주 걸리는 자리를 모았다. 진단 코드의 뜻은 부록 B 에, 설명은 표의 장에 있다.

겉모습

실수	진단	고치는 법
`output` 을 `input` 앞에 적는다	E-CLAUSE-ORDER	절 차례표를 따른다. 입력 밖의 절은 `--fmt` 가 옮긴다(3장)
`struct p .` 처럼 블록을 점이나 줄바꿈으로 연다	E-STMT-NODO	`struct p do … end` (`--fmt` 가 고친다)
`;` 나 `,` 로 닫는다	E-VOCAB-REMOVED	떨어진 마침표 `.`
`p.x` 로 칸을 읽는다	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`
매개변수와 같은 이름의 지역을 짓는다	E-NAME-SHADOW	새 이름을 짓는다
`count`·`text`·`len` 같은 이름을 쓴다	E-NAME-BUILTIN 따위	내장 연산·이음글 낱말은 이름이 될 수 없다
모듈 이름과 op 이름이 같다	E-NAME-DUP	모듈 이름을 바꾼다
`expr a lt b and b lt c`	E-TYPE-LOGICAL	`expr (a lt b) and (b lt c)`(8장)
`type pct be u8 .`	E-TYPE-DECL	`type pct u8 .`

표 50.1 — 겉모습에서 걸리는 실수

값과 흐름

실수	진단	고치는 법
`let` 에 `set`	E-IMMUTABLE	`var` 로 짓는다
정수를 조건에 쓴다	E-TYPE-COND	`gt n 0` 처럼 묻는 것을 적는다
넘침을 C 처럼 감길 것이라 기대한다	E-VM-OVERFLOW(실행 중)	`wrap_add`·`sat_add`·`chk_add` 로 처분을 고른다
`narrow u8 x` 가 잘라 줄 것이라 기대한다	E-VM-CAST(실행 중)	`narrow_wrap`·`narrow_sat`·`narrow_try`
`guard` 의 `else` 가 떠나지 않는다	E-GUARD-FALLTHROUGH	`return`·`break`·`continue`·`panic`, 아니면 `if`
어떤 길에서 `return` 이 없다	E-RETURN-PARTIAL	모든 길에서 돌려준다
`if` 를 값으로 쓴다	E-IF-VALUE	`var` 를 짓고 갈래마다 `set`
`match` 에서 경우를 빠뜨린다	E-MATCH-INEXHAUSTIVE	빠진 `case` 나 `case _ .`
열거의 갈래를 점 없이 줄마다 적는다	E-ENUM-DOT	갈래마다 `red .`
확인 없이 `some_value` 로 꺼낸다	E-VM-NONE(실행 중)	`guard is_some` · `value_or` · `match`
`array u8 4`	E-TYPE-ARRAY	`array 4 u8`

표 50.2 — 값과 흐름에서 걸리는 실수

효과·권한·메모리

실수	진단	고치는 법
`fn` 에서 출력한다	`E-EFFECT-CALC`	`proc` 과 `effects io`
`fn` 에 `effects none` 을 적는다	`E-EFFECT-REDUNDANT`	절을 지운다
`fn` 이 `mut` 매개변수로 호출자에게 쓴다	`E-EFFECT-PURITY`	`proc` 으로
`effects io` 인데 권한을 받지 않았다	`E-EFFECT-NO-CAP`	`input out cap io .`
권한을 받아 두고 내장 연산에 넘기지 않는다	`E-CAP-MISSING`	`write_out out 1 …` 처럼 첫 피연산자로
시작점이 데이터를 받는다	`E-ENTRY-PARAMS`	시작점은 권한만 받는다. 인자는 `cap args`
`mut` 없는 슬라이스에 쓴다	`E-TYPE-MUT`	`mut slice`
`ref` 로 쓴다	`E-TYPE-REF`	`mut_ref`
같은 값을 쓰기로 두 번 빌린다	`E-EXCL`	빌림을 나누거나 차례를 나눈다
지역의 참조를 돌려준다	`E-ESCAPE`	값을 돌려주거나 호출자의 저장소에 담는다
`mut ref slice`	`E-MREF-SLICE`	새 슬라이스를 돌려준다
영역의 바이트를 밖으로 들고 나간다	`E-REGION-ESCAPE`	영역을 넓히거나 값만 들고 나간다
파일·버퍼를 닫지 않는다	`E-OWN-INCOMPLETE`	`close`·`finish` 를 부른다. 버리려면 `drop`
옮긴 값을 다시 쓴다	`E-OWN-MOVED`	넘기기 전에 쓰거나 돌려받는다
운영체제 없는 대상에서 힙을 쓴다	`E-HEAP-NOHOST`	고정 창(`cap allocator`)과 `alloc`

표 50.3 — 효과·권한·메모리에서 걸리는 실수

추상과 동시성

실수	진단	고치는 법
트레이트 서명에 `fn`·`proc` 을 적는다	`E-TRAIT-SIG`	`area input s self . output u64 .`
효과 줄 없는 `proc` 으로 순수 서명을 갖춘다	`E-TRAIT-EFFECT`	`fn` 으로, 또는 `effects` 를 적는다
타입 인자가 트레이트를 갖추지 않았다	`E-BOUND-UNSAT`	`satisfies` 와 `op` 을 더한다
`comptime` 자리에 실행 값을 준다	`E-COMPTIME-ARG`	리터럴이나 모듈 상수
`task_group` 밖에서 `spawn <op>`	`E-SPAWN-SCOPE`	`task_group do … end` 로 감싼다
나누는 되풀이에서 누적 변수에 쓴다	`E-PAR-CARRY`	`reduce <자리> <연산> .`
`pipe` 에 정렬 같은 스테이지를 쓴다	`E-PIPE-STAGE`	스테이지 일곱과 종결자 다섯만 쓴다
효과 있는 붙은 `op` 을 `fn` 안에서 `method` 로 부른다	(이 판에서는 거절되지 않는다)	직접 `<타입>.<이름>` 으로 부른다(23장)

표 50.4 — 추상과 동시성에서 걸리는 실수

부록 D — 문법 요약

규범은 명세의 본문이고, 이 부록은 자주 찾는 모양을 모은 참고 자료다. 완전한 형식 문법은 아직 없다.

op 머리의 절 차례

```
[export] [unsafe] [extern] fn¦proc <이름>
  satisfies · lowdoc                       ① 무엇인가
  vector · priority                        ② op 전체의 성격
  input <이름> comptime …                  ③ 번역 시점 입력
  input <이름> cap … · region …            ④ 권한·영역 입력
  using <이름> <타입> .                    ⑤ 깎아 쓰는 할당기
  input <이름> <타입> .                    ⑥ 데이터 입력
  output <타입> .                          ⑦
  effects <원자…> [via <타입>] .           ⑧
  link "<이름>" · variadic · asm <기계>    ⑨ 바깥과 잇는 자리
  access · inplace <쓰기> <읽기> · invalidates <입력> · parallel <이름> split · reduce  ⑩ 메모리를 만지는 법 · 나누어 도는 방법
  requires [static¦debug¦assume] <조건> .  ⑪ 입력 조건
  ensures <조건> .                         ⑫ 출력 약속 (ret)
  errors <갈래> [<조건>] .                 ⑬ 실패
  tests … · schedule …                     ⑭
do
  <문장들>
end
```

몸이 C 에 있는 `extern op` 은 절을 같은 차례로 `do … end` **안에** 담는다 — `unsafe extern proc <이름> do <절>* end`. 블록에는 절만 온다.

선언

```
module <이름> .
use <모듈> [from "<자리>"] .
type <이름> <타입> .
newtype <이름> <타입> .
struct <이름> do [satisfies <트레이트> .] [layout packed .] [mmio <주소> .] <칸 이름> <타입> [big¦little] [rw¦ro¦wo] . … end
enum <이름> do <갈래> [<칸> <타입>]… . … end
trait <이름> do <op 이름> <절…> . … end
contract <이름> do requires <조건> . … end
actor <이름> do [satisfies …] state do <칸> <타입> . … end [failure restart max <수> .] [mailbox bounded <수> .] <op>… end
build profile <이름> .        build tier t0¦t1¦t2¦t3 .        build <모드> .
build option <이름> bool¦int¦choice … default <값> .
test <이름> [schedule explore_interleavings [limit <수>] .] do expect <조건> . … end
```

문장

```
let <이름> [<타입>] [using <출처>] be <식> .
var <이름> [<타입>] be <식> .
set <자리> <식> .                          (자리 = 이름 · field … · index …)
if <조건> . do … end [else do … end] .
while <조건> . do … end
```

```
for <이름> <슬라이스> do … end
guard <조건> . else <떠나는 문장> .
match <값> do case <패턴> [when <조건>] . do … end … end
return [<식>] .    break .    continue .    panic "<글>" .
drop <이름> .
region <이름> stack¦frame¦arena¦static¦heap¦mmap¦disk¦device do … end
task_group [cancel_on_error] do … end
pipe <원천> do <스테이지> . … <종결자> . end
spawn send <액터> <메시지> <값>… .    drain <액터> .    schedule .
```

패턴

```
case _ .                    나머지 전부
case <정수> .  case <아래> to <위> .
case <갈래> [<이름>…] .     case <패턴> or <패턴> .
case some <이름> .  case none .  case ok <패턴> .  case error .
case <이름> when <조건> .
```

식

```
<연산> <인자>…                          전위 --- 우선순위 없음, 폼 안의 폼은 괄호
expr <a> + <b> * <c>                    중위 섬 --- * / 가 + - 보다, 비교가 그 아래, and 가 or 보다 강함
make <타입> do <칸> <식> . … end        <열거>.<갈래> <값>…
field <값> <마디>…   index <슬라이스> <번호>   method <값> <이름> <인자>…
some <값>   ok <값>   error <갈래>   none
try <식> [else_none ¦ else_error <갈래>]
comptime <식>   size_of <타입>   config <이름>
spawn actor <타입>   send <액터> <메시지> <값>…   spawn <op> <인자>…   await <핸들>
alloc_bytes <뿌리> capacity <수>
```

머리와 닫개

폼은 머리 하나로 시작해 닫개 하나로 끝난다. 닫개는 떨어진 마침표 . 이고,) 는 안에 열린 것을 함께 닫는다. do … end 는 짝인 괄호라 end 는 자기 do 만 닫는다 — 블록을 몸으로 갖는 구문은 end 에서 끝나고, 블록을 값으로 쓰는 문장은 자기 . 으로 끝난다. 개행은 닫개가 아니라 공백이다. 괄호는 블록 경계를 넘지 못한다.

부록 E — 표준 라이브러리 모듈

제9부(32–37장)는 표준 라이브러리를 쓰임새별로 둘러보았다. 이 부록은 모듈마다 한 쪽씩, 무엇을 하고 어떤 op 을 내며 무엇을 하지 않는지를 모은 **참조**다. 처음부터 읽는 곳이 아니라 필요할 때 펴 보는 곳이다.

모듈 쪽은 같은 모양이다. 머리에 소스 파일 · 층 · 받는 권한을 적고, 무엇을 하는 모듈인지, op 한눈에, 쓰는 법, 설계와 경계, 짓지 않은 것 차례로 이어진다. 모듈마다 적는 성숙도(`experimental` · `incubating` · `standard`)는 소스 머리와 개발 저장소의 장부가 권위이고 판마다 움직이므로 여기 옮기지 않는다. 모든 모듈이 따르는 규약 — 호출자의 버퍼, 값으로 돌아오는 실패, 전량 아니면 무, 갚아야 하는 소유 — 은 32장에 있다.

모듈 이름은 파일 이름과 다를 수 있다. `use` 가 찾는 것은 파일 안의 `module` 선언이다(`lib/str.low` 는 `strings`, `lib/alloc.low` 는 `allocs`). 크로스 모듈 이름은 늘 한정해서 부르고(`strings.find`), 자주 쓰면 `use codec as c .` 로 별칭을 준다.

모듈	한 줄
`strings`	문자열 뷰 연산 — 자르고 찾고 비교한다
`strbuf`	소유 문자열 버퍼와 널 종단 `cstr`
`fmt`	호출자의 버퍼에 조립하는 포매팅
`utf8`	UTF-8 코드포인트 순회와 검증
`utf16`	UTF-16 서로게이트(surrogate) 산술
`unicode`	유니코드 속성 표 — 기계적으로 뽑았다
`codec`	16진 · base64
`regex`	역추적하지 않는 Pike VM 정규식
`term`	터미널 렌더러의 순수한 절반 — ANSI 조립 · 화면 diff · 폭
`sortlib`	`u64` 제자리 quicksort
`sortgen`	제네릭 정렬 — 비교는 타입이 들고 온다
`searchlib`	정렬된 슬라이스의 이진 탐색
`hashmap`	`u64 → u64` 오픈 어드레싱
`strmap`	바이트열 → `u64` 해시맵
`vecs`	자라는 바이트 벡터
`spsc`	락 없는 단일 생산자·단일 소비자 링 버퍼
`hash`	해시맵 자리(FNV-1a) · 손상 검출(CRC-32) · SHA-256
`math`	부동소수 수학 — `==` 대신 `close`
`random`	재현되는 열과 운영체제 엔트로피를 이름으로 가른다
`hmac`	HMAC-SHA256 · HKDF — 누가 보냈는가
`chacha`	ChaCha20 스트림 — 혼자서는 안전하지 않다
`poly`	Poly1305 — 키는 메시지마다 새것
`aead`	ChaCha20-Poly1305 봉인과 개봉 — 암호는 여기서 시작한다
`x25519`	키 합의
`aes`	AES-128 블록 암호 — 혼자 쓰면 대개 틀린다

gcm	AES-128-GCM — 논스를 되풀이하지 않는다
bigint	큰수 모듈러 산술
rsa	RSASSA-PSS 검증
p256	NIST P-256 곡선과 ECDSA 검증
p384	NIST P-384 곡선과 ECDSA 검증 — 공개 웹의 중간 인증기관
ecdsa	ECDSA P-256 서명 — 난수 없이 논스를 유도한다
ed25519	Ed25519 서명 검증
der	DER 최소 파서 — 공개키만 꺼낸다
pem	PEM 봉투 벗기기
x509	X.509 인증서 읽기 — 자리와 길이만 낸다
verify	인증서 서명과 체인의 한 마디 확인
tls13	TLS 1.3 의 부품 — 키 스케줄 · 레코드 · 전사 · Finished
tlssrv	서버 TLS 핸드셰이크와 응용 데이터 레코드
tlscli	클라이언트 TLS 1.3 핸드셰이크 — 인증서 확인은 부르는 쪽이 한다
http	HTTP/1.1 요청 파서 — 알맹이는 거절이다
soa	구조체 배열을 칸별 배열로 두는 배치의 시범

표 50.1 — L0 — 순수 계산(effects none · 호출자의 버퍼)

모듈	한 줄
allocs	할당기 트레이트 · 범프 할당기 · 기본 할당기 둘
pool	세대 핸들 블록 풀
shard	저장소를 겹치지 않는 조각으로 나누는 토큰
budget	핸들의 비트 예산과 세대 수의 은퇴
wire	한 워드를 칸으로 나눠 쓰기 — 마스크가 자리와 폭을 든다
flags	이름 붙은 켬/끔 설정을 한 워드에
segarena	고정 크기 세그먼트 아레나
pagecache	페이지 번호와 고정 토큰
growvec	자라는 바이트 벡터 — vecgen.vec u8 의 짧은 이름
vecgen	제네릭 자라는 벡터 vec t
mapgen	제네릭 해시맵 table k v
nodelist	고정 크기 침입형 목록
segview	흩어진 조각 뷰의 커서 · 총길이 · 펴기
lifemode	lifemode · lifeatom — 값이 언제 끝나는가

표 50.2 — L1 — 저장(할당 권한이나 빌린 바이트가 필요하다)

모듈	한 줄
io	슬라이스 위의 스트림 읽기
outbuf	버퍼링 출력 — 비우기를 잊으면 번역이 거절한다
files	파일과 디렉터리 — 닫기를 잊으면 번역이 거절한다
tty	터미널 입력 — raw 모드 · 키 읽기 · 순수 키 파싱

`net`	소켓 — TCP 루프백 · 프로세스 안 연결 쌍
`clock`	시각과 마감 — 단조 시계와 벽시계
`trust`	신뢰 저장소에서 믿는 뿌리 찾기 — 묶음을 흘려 읽는다

표 50.3 — L2 — 호스트(권한이 필요하다)

암호 모듈은 TLS 1.3 을 향해 층으로 쌓여 있고, 그 차례와 아직 없는 꼭대기는 36장에 있다.

strings — 문자열 뷰 연산

소스 lib/str.low
층 L0 — 순수 계산
권한 없음

문자열을 **복사하지 않고** 비교하고, 찾고, 다듬고, 쪼갠다. 입력을 공백으로 나누거나, 접두사를 떼거나, 부분 문자열을 찾을 때 쓴다. 결과는 언제나 원본 위의 **뷰** — 남의 바이트를 가리키기만 하는 창 — 이고, 새 메모리가 한 바이트도 들지 않는다.

```
use strings .

if strings.starts_with line "GET " . do
  let rest slice u8 be strings.remove_prefix line "GET " .
end
```

rest 는 line 의 일부를 그대로 가리킨다. 이 모듈은 언어가 이미 가진 것(len·index·subslice·eq·option·guard·while·actor)만으로 지어졌고 내장 연산을 하나도 늘리지 않았다. 문자열 연산의 대부분은 소유가 필요 없다 — 자르기·찾기·비교는 뷰를 낼 뿐이다. 이어 붙이기처럼 쓰기 가능한 버퍼가 필요한 일은 strbuf 가 맡는다.

설계와 경계

- **하는 것** — 바이트 단위 비교·탐색·접두와 접미·다듬기·분할. 모든 fn 이 effects none 이고 아무것도 할당하지 않는다.
- **인코딩을 해석하지 않는다.** str 은 바이트다. UTF-8 검증과 해독은 utf8 의 일이다.
- **소유와 수정을 하지 않는다.** 원본을 한 바이트도 바꾸지 않는다.
- **str_len·str_sub 는 없다.** str 은 slice u8 의 별칭이므로 내장 len·subslice 가 그대로 돈다. 같은 뜻에 두 이름을 두지 않는다.
- **자리표 값이 없다.** "−1 이면 못 찾음" 같은 관례를 쓰지 않는다. 못 찾음은 option 의 none 이다.

type str slice u8 . 은 **투명 별칭**이다. newtype 이 아니므로 문자열 리터럴, slice u8 값, subslice 결과가 모두 그대로 str 자리에 들어간다. 상태를 드는 것은 str_splitter 액터 하나뿐이다 — 원본 src, 구분 바이트 sep, 커서 pos, 소진 여부 fin.

op 한눈에

모든 fn 은 effects none 이고, 액터의 proc 둘은 effects state 다. 부를 때는 strings.find 처럼 모듈 이름을 붙인다.

op	시그니처	안 될 때
eq_str	(a str, b str) → bool	다르면 false
has_byte	(s str, b u8) → bool	없으면 false
find	(hay str, needle str, from u64) → option u64	못 찾거나 from > len hay 이면 none
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	접두사가 없으면 **원본 그대로**
remove_suffix	(s str, suffix str) → str	접미사가 없으면 원본 그대로
trim_start	(s str, cut str) → str	모두 잘리면 빈 뷰
trim_end	(s str, cut str) → str	모두 잘리면 빈 뷰
split_next	(src str, sep u8, pos u64) → option str	pos > len src 이면 none
str_splitter	액터: init(s2 slice u8, b u8) → u64 · next → option slice u8	소진 뒤 next 는 none

표 50.1 — strings 의 op

op 상세

넘기는 slice u8 은 뷰이고 돌려받는 것도 같은 바이트 위의 다른 뷰다. 그래서 mut 가 아니고 effects none 이며, 원본이 살아 있는 동안만 유효하다. 비교와 검색의 두 인자는 늘 (찾을 대상, 찾는 것) 차례다 — has_byte s b 는 "s 안에 b 가 있는가" 로 읽는다.

eq_str — 바이트 단위 동등 비교. 길이가 다르면 바이트를 보기도 전에 false 다. 내장 eq 는 스칼라 전용이라 슬라이스 비교가 여기 산다. 그래서 이름이 eq 로 짧아지지 못했다.

has_byte — 바이트 b 하나가 s 안에 있는가. 문자열이 아니라 바이트 하나다. trim_start·trim_end 의 잘라낼 집합 검사가 이것을 쓴다.

find — hay 에서 needle 을 from 부터 찾아 시작 색인을 some 으로 낸다. from 이 있는 이유가 이 모듈의 성격이다. **커서는 호출자가 드는 값이다.** 다음 검색은 찾은 자리 뒤를 from 으로 다시 부른다. 라이브러리가 상태를 어디에도 숨기지 않으므로, 같은 문자열을 여러 곳에서 동시에 훑어도 서로 간섭하지 않는다. from > len hay 면 none, 빈 needle 은 some from 이다.

has — "들어 있는가" 만 물을 때. 속은 is_some (find hay needle 0) 그대로다.

starts_with · ends_with — 접두와 접미 검사. 조각이 s 보다 길면 볼 것도 없이 false 다.

remove_prefix · remove_suffix — 접두나 접미를 뗀 뷰. 떼어 낼 조각이 **없으면 원본을 그대로** 돌려준다. 실패가 아니므로 반환이 option 이 아니다. 떼였는지 알아야 하면 먼저 starts_with·ends_with 로 묻는다.

trim_start · trim_end — 앞이나 뒤에서 특정 바이트들을 걷어낸 뷰. cut 은 차례 있는 문자열이 아니라 **바이트의 집합**이다 — " \t" 는 "공백 다음 탭" 이 아니라 "공백 또는 탭" 이다. 모두 잘리면 길이 0 인 뷰가 나온다.

split_next — src 를 구분 바이트 sep 로 쪼갤 때, pos 부터 다음 sep 앞까지의 조각을 some 으로 낸다. **커서 전진은 호출자 몫이다** — 조각을 받으면 pos ← pos + len(조각) + 1 로 스스로 민다(+1 은 구분 바이트 한 칸). 뒤에 sep 이 없는 마지막 조각도 some 이고, pos > len src 면 none 이 끝 신호다. "a,b" 는 "a"·"b", "a," 는 "a"·""(꼬리 빈 조각), "" 는 "" 하나다.

str_splitter — split_next 의 당겨 읽기 판. 커서를 액터 상태가 든다. init 에 원본과 구분 바이트를 넣으면 이후 next 는 인자가 없고, 조각을 option 으로 내다가 소진되면 none 이다.

쓰는 법

모듈 이름은 파일 안의 module 선언이다. lib/str.low 의 모듈은 strings 다.

```
use strings from "lib/str.low" .   rem 경로를 직접 적는다(부르는 파일에서의 상대 경로)
use strings .                      rem 표준 모듈 자리에서 풀린다
use strings as s .                 rem 별칭 --- 이후 s.eq_str 로 부른다
```

찾고 쪼갠다.

```
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" 를 ',' (바이트 44) 로: "aa"·"b"·""·"cc" --- 네 조각
  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` 는 6 을, `t_split` 은 4 를 낸다. 상태를 원하면 액터 판을 쓴다 — 커서를 액터가 든다.

```
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
```

두 판의 차이는 커서를 누가 드는가 하나다. 같은 원본을 여러 갈래로 동시에 훑을 일이 있으면 `split_next` 가 맞고, 반복 하나로 끝까지 당길 뿐이면 액터 판이 짧다.

반례

반례. 내장 `eq` 로 슬라이스를 비교한다

```
guard eq "abc" "abc" . else return 0 .        rem ✗ eq 는 스칼라 전용이다
```

`E-VM-TYPE` 이다. 슬라이스 비교는 `strings.eq_str` 로 한다.

반례. 분할 반복에서 커서를 밀지 않는다

```
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 가 그대로
end
```

번역 오류가 아니다 — **프로그램이 끝나지 않는다.** 같은 `pos` 를 주므로 같은 조각이 영원히 나온다. 조각을 받을 때마다 `pos + len(조각) + 1` 로 민다.

반례. `option` 을 값처럼 쓴다

```
let r option u64 . be strings.find "abc" "zz" 0 .
return some_value r .                       rem ✗ none 검사가 없다
```

번역은 통과하고, 못 찾은 순간 실행 중 `E-VM-NONE` 으로 멈춘다. 찾히는 입력으로만 시험하면 드러나지 않는다. `guard is_some r . else …` 가 먼저다 (11장).

반례. 뷰가 가리키는 원본을 나중에 바꾼다

```
let piece slice u8 be strings.remove_prefix line "GET " .
set (index line 4) 88 .                                  rem ✗ 원본을 고쳤다
```

오류 없이 `piece` 의 내용이 조용히 바뀐다 — 복사본이 아니라 창이기 때문이다. 내용을 붙잡아 두려면 `strbuf` 로 복사한다.

주의

- **바이트 단위다, 글자 단위가 아니다.** 한글이나 이모지처럼 여러 바이트로 된 문자는 한 단위가 아니다. `needle` 로는 바이트열 그대로 찾히지만, `cut` 은 바이트 집합이라 멀티바이트 문자를 한 단위로 다루지 못한다. 글자 단위는 `utf8` 로 간다.
- **뷰는 원본과 수명과 내용을 함께한다.** 원본이 바뀌면 뷰의 내용도 바뀌고, 원본이 사라지면 뷰도 못 쓴다. “값을 받았다” 가 아니라 “창을 받았다” 로 읽는다.
- **`remove_prefix` 는 실패를 알리지 않는다.** 뗐는지가 필요하면 `starts_with` 로 먼저 묻는다.
- **`find` 는 소박한 탐색이다.** 비용은 $O(\texttt{len hay} \times \texttt{len needle})$ 이다. 아주 긴 문자열을 되풀이해 훑는 자리에서는 이 비용을 계산에 넣는다.
- **커서 판과 액터 판.** `split_next` 는 상태를 나누지 않으므로 여러 커서가 동시에 읽어도 자유롭다. `str_splitter` 는 상태를 들므로 띄운 곳이 소진까지 책임진다.
- **널 종단 문자열로 바로 가는 길은 없다.** 뷰는 끝의 0 바이트를 약속하지 못한다. `strbuf` 의 `as_cstr` 를 거친다.
- **내장 이름과 겹치는 이름.** `eq·contains` 는 내장 연산이라 이 모듈이 `eq_str·has_byte` 로 이름을 지었다. 내 코드의 변수 이름도 내장 이름을 피한다.

strbuf — 소유 문자열 버퍼와 널 종단 cstr

소스 lib/strbuf.low
층 L0 — 순수 계산(호출자의 버퍼)
권한 없음 · as_cstr 만 unsafe

문자열을 **조금씩 이어 붙여** 만든다. 디렉터리와 "/" 와 이름으로 경로를 조립하거나, 메시지를 여러 조각으로 만들 때 쓴다. strings 가 뷰(읽기)라면 strbuf 는 쓰기다 — 이어 붙이고, 되는 만큼 넣고, 키운다.

```
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 .
```

이 모듈에서 "버퍼" 는 두 조각이다 — 얼마나 썼는지 기억하는 **상태** b 와 실제 바이트가 들어가는 **자리** buf. 둘은 딴 몸이므로 op 을 부를 때 늘 함께 넘긴다. 버퍼를 어디서 얻을지(cap allocator·영역·정적 버퍼)는 호출자의 일이고, 라이브러리는 **몰래 할당하지 않는다**. 늘어난 내장 연산은 슬라이스의 생 포인터를 내는 cstr_of 하나다.

설계와 경계

자리가 모자랄 때 무엇을 할지는 호출자가 이름으로 고른다. wrap_add·sat_add·chk_add 와 같은 규율이다. 한 op 이 상황 봐서 알아서 하지 않는다.

op	갈래	행동
append	전량 아니면 무	안 들어가면 **한 바이트도 쓰지 않고** 거절한다
append_trunc	되는 만큼	쓴 개수를 낸다 — 잘렸다는 사실이 **값**으로 나온다
append_grow	키운다	더 큰 자리를 **호출자가 준다**

표 50.1 — 이어 붙이기의 세 갈래

- **바이트를 소유하지 않는다.** str_buf 는 상태(len)만 들고, 바이트 버퍼는 매 호출마다 인자로 받는다. 용량을 두 곳에 적으면 어긋나므로 구조체에 없다 — 호출자가 준 buf 의 len 이 곧 용량이다.
- **저절로 자라지 않는다.** 성장은 append_grow 에 더 큰 버퍼를 호출자가 주는 것으로만 표현된다.
- **뷰에서 cstr 로 바로 가는 길이 없다.** 뷰는 끝의 0 바이트를 약속하지 못한다. cstr 를 만드는 유일한 안전한 길이 str_buf 를 거치는 것이고, 그 경유가 할당을 보이게 한다.

> **널 봉인**
>
> 모든 이어 붙이기는 **1 바이트를 남겨 둔다** — 내용 용량은 len(buf) - 1 이다. 쓴 뒤 buf[len] = 0 으로 봉인하므로 as_cstr 가 O(1)·무할당이다. 버퍼는 최소 1 바이트여야 하고, 실용적으로는 **내용 n 바이트에 버퍼 n + 1 바이트**로 외워 둔다.

타입은 셋이다. str_buf(상태 len u64), sb_error(실패 이름 no_room 하나 — 실패는 멈춤이 아니라 값이다), cstr(널 종단 char* 를 가리키는 생 포인터의 newtype, 다른 언어와 주고받는 경계 전용).

op 한눈에

op	시그니처	안 될 때
new	fn () → str_buf	—
as_str	fn (b str_buf, buf slice u8) → slice u8	— (뷰, 복사가 아니다)
room	fn (b str_buf, buf slice u8) → u64	빈 버퍼(길이 0)면 넘침으로 멈춘다
as_cstr	unsafe proc (b str_buf, buf mut slice u8) → cstr	len + 1 > len(buf) 면 경계 밖으로 멈춘다
append	proc (b mut str_buf, buf mut slice u8, s slice u8) → result void sb_error	error no_room(버퍼 그대로)
append_trunc	proc (b mut str_buf, buf mut slice u8, s slice u8) → u64	— (쓴 개수 < len s 로 안다)
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(원본 그대로)

표 50.2 — strbuf 의 op

op 상세

- new — len 0 인 빈 상태. 매개변수가 없다 — 만드는 것은 상태뿐이다.
- as_str — 지금까지 쓴 바이트의 뷰, 곧 subslice buf 0 len. 널은 뷰에 들지 않는다 — 길이가 진실이다.
- room — 남은 내용 용량 len(buf) - 1 - len. 널 자리 1 바이트가 빠져 있다. 이어 붙이기 전에 "이만큼 더 넣을 수 있나" 를 물을 때 쓴다.
- as_cstr — len 자리에 널을 봉인하고 시작 포인터를 cstr 로 낸다. buf 가 mut 인 까닭은 널 1 바이트를 **실제로 쓰기** 때문이다. cstr_of 가 effects unsafe 라 이 op 도 unsafe 이고, 순수 코드에서는 못 부른다.
- append — len + len(s) + 1 ≤ len(buf) 면 모두 쓰고 봉인하고 ok. 아니면 error no_room 이고 **한 바이트도 쓰지 않는다**(길이도 버퍼도 그대로). 절반 쓰인 버퍼는 조용히 틀린 내용이고, 그것은 거절보다 나쁘다.
- append_trunc — 남은 자리만큼만 쓰고 쓴 개수를 돌려준다. 그 수를 len s 와 비교하면 잘렸는지 안다. 비교하지 않으면 잘린 줄 모른다.
- append_grow — 옛 내용을 new_buf 로 옮기고 s 를 잇고 봉인하는 것까지가 한 동작이다. new_buf 매개변수가 곧 "몰래 할당하지 않는다" 의 실물이다. 새 자리조차 모자라면 error no_room 이고 **아무것도 옮기지 않는다**. 성장하려면 더 큰 바이트를 가진 자여야 한다.

쓰는 법

세 갈래가 서로 다르게 행동한다. 버퍼는 호출자가 마련한다.

```
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 봉인 확인 --- len 자리(buf[2])에 널이 있다
  guard eq (index buf 2) 0 . else return 92 .
  rem 버퍼 5 = 내용 4 + 널 1 --- 2 + 3 + 1 > 5 이므로 거절되고 버퍼가 더럽혀지지 않는다
```

```
  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 같은 상황에서 trunc 는 되는 만큼(2) 쓰고 2 를 돌려준다
  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
```

성장은 새 자리를 호출자가 주는 것으로 표현된다.

```
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 더 큰 자리를 내가 마련해 넘긴다 --- 라이브러리는 할당하지 않는다
  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` 는 4(“ab” 2 + 되는 만큼 쓴 2)를, `grow` 는 5(“abxyz”)를 낸다. 실제 쓰임은 경로 조립이다 — `append b pathbuf dpath`, `append b pathbuf "/"`, `append b pathbuf name` 으로 잇고 `as_str b pathbuf` 로 뷰를 꺼내 `files` 에 넘긴다.

반례

반례. 순수 코드에서 `as_cstr` 를 부른다

```
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 ✗ unsafe proc 을 fn 에서 부른다
  return 0 .
end
```

`E-EFFECT-CALC` 로 번역에서 거절된다. `cstr` 를 만드는 것은 널 종단의 보증이고, 그 보증은 `unsafe` 경계 안에서만 선다. 부르는 쪽을 `unsafe proc` 으로 바꾼다.

반례. 널 자리를 잊은 버퍼 크기

버퍼가 정확히 2 바이트면 `"ab"` 조차 들어가지 않는다(2 + 0 + 널 1 > 2). 첫 `append` 부터 `error no_room` 이고 `field b len` 이 0 에서 움직이지 않는다. 내용 n 바이트에는 버퍼 n + 1 바이트가 필요하다.

반례. `result` 를 보지 않고 버퍼를 읽는다

```
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 을 보지 않았다
```

멈춤도 오류도 없다. 버퍼가 깨진 것도 아니다 — 거절은 버퍼를 건드리지 않는다. 다만 **기대한 것이 안 들어간 짧은 문자열**이 나온다. `guard is_ok r . else …` 가 먼저다.

반례. 상태와 버퍼를 섞어 넘긴다

```
let r result void strbuf.sb_error . be strbuf.append b buf1 "ab" .
let v slice u8 be strbuf.as_str b buf2 .   rem ✗ 길이는 buf1 의 것, 바이트는 buf2 의 것
```

오류 없이 엉뚱한 바이트의 뷰가 나온다. 하나의 `str_buf` 에는 하나의 버퍼(또는 `append_grow` 로 옮긴 그 후계)만 짝지어 쓴다.

주의

- **버퍼는 내용보다 1 바이트 크게 잡는다.** 널 봉인 자리 때문이다.
- `append` **의 거절은 조용하다.** 멈춤이 아니라 값이므로 `result` 를 보지 않으면 프로그램이 그냥 계속 돈다.
- **성장한 뒤에는 옛 버퍼를 버린다.** `append_grow` 가 `ok` 를 낸 순간 진실은 `new_buf` 에 있다. 이후 호출에 `old` 를 넘기지 않는다.
- `as_cstr` **는 스냅숏이 아니다.** 시작 포인터를 그대로 내므로 그 뒤 이어 붙이면 가리키는 내용이 바뀐다. 다른 언어를 부르기 직전에 만들고 바로 쓴다.
- `room` **은 길이 0 인 버퍼에서 멈춘다** (`0 - 1`). 버퍼는 최소 1 바이트다.

fmt — 호출자의 버퍼에 조립하는 포매팅

소스 lib/fmt.low
층 L0 — 순수 계산(호출자의 버퍼)
권한 없음

수와 글자를 **호출자가 준 버퍼에** 찍어 문자열을 만든다. 출력할 줄이나 로그 한 줄을 조립할 때 쓴다. 이 모듈은 종이를 만들지 않고 건네받은 종이 (mut slice u8)에 글씨만 쓴다.

```
use fmt .

let p option u64 . be fmt.put_u64 buf 0 1234 .
guard is_some p . else return 1 .
```

0 은 pos(다음에 쓸 자리)이고, 돌려받는 option u64 는 **성공하면 새 pos, 자리가 모자라면** none 이다.

포매팅은 출력이 아니다. 모든 op 이 호출자의 버퍼에 바이트를 조립할 뿐이고 하나의 예외 없이 effects none 이다. 실제 출력은 cap io 를 든 호출자가 write_out 이나 outbuf 로 따로 한다. 그래서 조립 코드는 권한 없이 아무 데서나 부를 수 있고, 시험은 화면이 아니라 버퍼를 대조하면 되며, 무엇을 언제 내보낼지는 호출자가 정한다. 수를 바이트로 바꾸는 일은 커널도 할당도 필요 없어 라이브러리다 — 내장 연산은 하나도 늘지 않았고, 파일 전체가 VM·네이티브 대조 안에 있다.

설계와 경계

- **규약은 하나다 — 모든 라이터는 (buf, pos) → 새 pos.** 성공하면 some 새pos, 모자라면 none 이다. 멈추지 않고 값으로 거절한다. 새 pos 를 그대로 다음 호출에 주면 글자가 이어 붙는다.
- **전량 아니면 무.** 자리가 모자라면 한 바이트도 쓰지 않는다 — "1234" 대신 "12" 가 남으면 아무도 알아채지 못한다. 그래서 put_u64·put_hex 는 자릿수를 먼저 세고(dec_width·hex_width) 앞에서부터 쓴다.
- **줄바꿈은 LF 하나다**(put_nl). io 의 take_line 과 같은 규약이고, CRLF 는 만들지도 지우지도 않는다.
- **하지 않는 것** — 부동소수 포매팅, 폭 맞춤과 정렬, 로케일(천 단위 구분). 필요하면 dec_width 로 폭을 재서 put_byte 로 빈칸을 직접 채운다.
- **상태가 없다.** 구조체도 열거도 없다. 버퍼와 커서는 모두 호출자의 것이라, 어디서 몇 개를 동시에 써도 서로 간섭하지 않는다.

op 한눈에

op	시그니처	안 될 때
put_byte	proc (buf mut slice u8, pos u64, b u8) → option u64	자리가 없으면 none, 버퍼 그대로
put_str	proc (buf, pos, s slice u8) → option u64	모두 들어가지 않으면 none, 버퍼 그대로
dec_width	fn (n u64) → u64	실패 없음
put_u64	proc (buf, pos, n u64) → option u64	자리가 없으면 none, 버퍼 그대로
put_i64	proc (buf, pos, n i64) → option u64	자리가 없으면 none
hex_width	fn (n u64) → u64	실패 없음
put_hex	proc (buf, pos, n u64) → option u64	자리가 없으면 none, 버퍼 그대로
put_bool	proc (buf, pos, b bool) → option u64	자리가 없으면 none
put_nl	proc (buf, pos) → option u64	자리가 없으면 none

표 50.1 — fmt 의 op — 모두 effects none

이름의 put_ 은 “버퍼에 써 넣는다”, _width 는 “몇 바이트가 필요한지 세기만 한다” 는 뜻이다. 버퍼를 고치는 것은 proc, 세기만 하는 것은 fn 이지만 아홉 모두 바깥에 닿지 않는다.

op 상세

모든 라이터의 앞 두 매개변수는 같다. buf 는 **어디에** 쓸지(할당하지 않으므로 받아야 하고, 고치므로 mut), pos 는 그 안 **어느 자리부터** 쓸지(커서를 모듈이 기억하지 않으므로 직전 호출이 준 새 pos 를 넘기는 것이 “이어 쓴다” 의 유일한 표현)다.

- **put_byte** — 바이트 b 하나를 buf[pos] 에 쓴다('-' 는 45, LF 는 10). 성공하면 some (pos+1).
- **put_str** — 바이트열 s 전체를 쓴다. s 는 읽기만 하므로 문자열 리터럴을 그대로 줄 수 있다. pos + len s 가 len buf 를 넘으면 none.
- **dec_width · hex_width** — n 을 10 진·16 진으로 쓸 때의 자릿수. 최소 1(0 도 한 자리). 전량 아니면 무를 위한 사전 계산이지만, “버퍼가 몇 바이트면 충분한가” 를 미리 잴 때 직접 써도 된다.
- **put_u64 · put_hex** — 부호 없는 10 진·16 진(소문자, 0x 없음). 0x 가 필요하면 put_str buf pos "0x" 를 앞에 붙인다.
- **put_i64** — 부호 있는 10 진. 음수면 - 를 먼저 쓰고 크기를 u64 에서 다룬다. i64 최솟값(−9223372036854775808)은 양수로 뒤집을 수 없으므로 0 - n 을 u64 에서 계산한다. 이 경계값도 옳게 찍힌다.
- **put_bool** — "true"·"false" 를 쓴다. "1"·"0" 이 아니므로 4 나 5 바이트가 필요하다.
- **put_nl** — LF 한 바이트. put_byte buf pos 10 과 같다.

쓰는 법

조립은 순수하게, 출력은 마지막에 cap io 로 한 번. 돌려받은 새 pos 를 **다음 호출에 이어서 넘기는 것**이 핵심이다.

```
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 조립 --- 여기까지 한 바이트도 밖으로 나가지 않았다
  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 출력 --- 조립된 앞부분(0 … p5)만 내보낸다
  let w u64 be write_out out 1 (subslice buf 0 (some_value p5)) .
  return 0 .
end
```

출력은 `answer=42 hex=ff` 와 줄바꿈 하나다. 조각마다 종료 코드를 달리 두면 어디서 모자랐는지 바로 안다. 버퍼 크기를 미리 재는 것도 같은 도구로 된다.

```
fn need_for input n u64 . output u64 . do
  let w u64 be fmt.dec_width n .
  return add (add 7 w) 1 .
end
```

반례

반례. `effects none` op 안에서 출력한다

```
proc bad input out cap io . output u64 . effects none . do
  return write_out out 1 "x" .
end
```

선언이 거짓말이 되어 `E-EFFECT`(`fn` 이면 `E-EFFECT-CALC`)로 거절된다. `effects io` 로 바꿔도 `cap io` 입력이 없으면 `E-EFFECT-NO-CAP` 이다.

반례. 돌려받은 `pos` 를 버리고 옛 `pos` 를 다시 쓴다

```
let p1 option u64 . be fmt.put_str buf 0 "answer=" .
let p2 option u64 . be fmt.put_u64 buf 0 42 .        rem ✗ pos 는 some_value p1 이어야 한다
```

번역은 통과하고 결과가 `answer=42` 대신 `42swer=` 처럼 겹쳐 나온다. 출력 앞부분이 뭉개져 있으면 `pos` 를 이어 넘겼는지부터 본다.

반례. `none` 을 무시하고 진행한다

자리가 모자라면 그 조각은 **아예 쓰이지 않은 것**이다. `guard is_some … else return …` 없이 진행하면 중간 조각이 빠진 채 나간다. 긴 값일 때만 출력이 이상하면 버퍼 부족이다. 전량 아니면 무는 op 하나의 보장이지 이어진 호출 전체의 보장이 아니다. 거꾸로 `guard` 없이 `some_value` 로 꺼내면, 자리가 모자란 순간 `E-VM-NONE` 으로 멈춘다 — 짧은 값으로만 시험하면 드러나지 않는다.

반례. `write_out` 에 버퍼 전체를 준다

`subslice buf 0 (some_value p5)` 대신 `buf` 를 주면 쓰지 않은 뒷부분까지 나간다. 줄 끝에 정체 모를 바이트가 붙는다. 마지막 `pos` 가 곧 **쓴 길이**다.

주의

- **모자람은 정상 경로다.** 오류가 아니라 예상된 답이므로 `guard` 로 받는다. 상한이 필요하면 미리 잰다 — `u64` 10 진 최대 20 자리, 16 진 최대 16 자리, `i64` 는 부호 1 자리를 더한다.
- **`pos` 는 색인이자 길이다.** 조립이 끝난 뒤의 `pos` 는 지금까지 쓴 바이트 수와 같아서 `subslice buf 0 pos` 가 곧 결과다.

- **버퍼를 다시 쓰려면 `pos` 를 0 으로 되돌리기만 하면 된다.** 지난 내용은 지울 필요가 없다 — 그때의 `pos` 까지만 내보내면 옛 바이트는 보이지 않는다.
- **CRLF 는 직접 쓴다.** 필요한 규약이면 `put_str buf pos "\r\n"` 을 쓴다.
- **모든 op 이 `effects none` 이다.** `fn` 안에서도 액터 핸들러 안에서도 부를 수 있다. 출력할 때만 `cap io` 와 `effects io` 가 필요하다.
- **접두사·부호·폭 맞춤은 호출자 몫이다.** `0x`·`+`·천 단위 쉼표·빈칸 채우기를 `put_str`·`put_byte` 로 직접 붙인다.

utf8 — UTF-8 코드포인트 순회와 검증

소스 lib/utf8.low
층 L0 — 순수 계산
권한 없음

UTF-8 바이트열을 **코드포인트 단위로** 훑고 검사한다. 코드포인트는 문자 하나에 유니코드가 붙인 번호다('한' = U+D55C = 54620). UTF-8 에서 한 코드포인트는 1 … 4 바이트를 먹는다 — 번호 하나가 바이트 하나가 아니다. 한글과 이모지가 섞인 문자열의 글자 수를 세거나, 입력이 올바른 UTF-8 인지 볼 때 쓴다.

```
use utf8 .

let c option u64 . be utf8.decode s 0 .
guard is_some c . else return 1 .
```

`0` 은 "첫 글자" 가 아니라 **바이트 0 번지**다. 이 모듈의 위치는 모두 바이트 오프셋이고, 글자 단위 전진은 `next` 가 한다.

왜 검증이 필요한가. UTF-8 은 가변 길이라 아무 바이트열이나 올바르지 않다. 시퀀스가 도중에 잘리거나, 이어지는 바이트가 엉뚱하거나, 같은 글자를 필요보다 긴 바이트로 적은 **과길이 인코딩**일 수 있다. 과길이가 특히 위험하다 — 같은 코드포인트를 두 바이트열로 쓸 수 있게 되면 "이 문자열에 `/` 가 있는가" 같은 필터가 거짓말을 한다. 보안 필터 우회의 고전적인 통로다.

왜 라이브러리인가. `str` 은 바이트이고(9장), UTF-8 을 타입 불변식으로 강제하지 않았다. 인코딩은 위층의 선택이고 이 모듈이 그 층이다. Rust 는 `str` 에 UTF-8 을 강제해 글자 순회를 공짜로 얻는 대신 모든 생성 경로에 검증을 물린다. 여기서는 검증을 **원할 때만** 치르고, 그 덕에 바이트 자르기 (`subslice`)는 실패할 수 없다. 내장 연산은 하나도 늘지 않았다.

설계와 경계

- **잘못된 UTF-8 은 값으로 답한다.** `none` 이 곧 "여기서부터 UTF-8 이 아니다" 다. 멈추지도, 치환 문자(U+FFFD)를 넣지도 않는다 — 치환은 되돌릴 수 없는 손실이라 호출자가 고를 수 없게 만든다.
- **거절하는 것** — 과길이 인코딩, 서로게이트(U+D800 … DFFF), U+10FFFF 초과, 잘린 시퀀스, 선두 자리의 이어지는 바이트. 서로게이트는 UTF-16 이 큰 문자를 두 칸에 나눠 담을 때 쓰는 부품이라 UTF-8 안에는 자리가 없다(`utf16`).
- **부분 답을 주지 않는다.** `count_chars` 는 무효 바이트를 만나면 `none` 이다.
- **짓지 않은 것** — 정규화, 로케일 비교, 글자 묶음(grapheme cluster, `term`), 대소문자 변환, 인코딩(코드포인트 → 바이트). 순회와 검증만이다.

코드포인트 범위	바이트 수	선두 바이트	이어지는 바이트
U+0000 … U+007F	1	`0xxxxxxx`	없음
U+0080 … U+07FF	2	`110xxxxx`	`10xxxxxx` × 1
U+0800 … U+FFFF	3	`1110xxxx`	`10xxxxxx` × 2
U+10000 … U+10FFFF	4	`11110xxx`	`10xxxxxx` × 3

표 50.1 — UTF-8 의 바이트 배치

표에서 세 가지가 나온다. 선두 바이트만 보면 길이를 안다(seq_len). 10xxxxxx 가 선두 자리에 오면 글자의 시작이 아니다. 길이마다 담을 수 있는 최소 코드포인트(2 바이트는 128 이상, 3 바이트는 2048 이상, 4 바이트는 65536 이상)보다 작은 값이 나오면 과길이다 — decode 가 이 문턱으로 거른다.

op 한눈에

op	시그니처	안 될 때
is_cont	(b u8) → bool	실패 없음
seq_len	(b u8) → u64	선두 바이트가 아니면 0
decode	(s slice u8, at u64) → option u64	무효·잘림·범위 밖이면 none
next	(s slice u8, at u64) → option u64	무효·잘림·끝이면 none
count_chars	(s slice u8) → option u64	무효 바이트가 하나라도 있으면 none
is_valid	(s slice u8) → bool	실패 없음

표 50.2 — utf8 의 op — 모두 fn · effects none

op 상세

이 모듈은 상태를 들지 않으므로 어디를 어디서부터 읽을지는 모두 호출자가 준다.

- is_cont — 이어지는 바이트인가(10xxxxxx, 곧 eq (bit_and b 192) 128). 앞뒤 문맥이 필요 없는 판정이라 바이트 하나를 받는다.
- seq_len — 선두 바이트로 그 코드포인트의 바이트 길이(1 … 4)를 안다. 길이 정보는 선두 바이트의 앞쪽 비트에만 있다. **0 은 길이가 아니라 오류 신호**다 — 그대로 더하면 커서가 움직이지 않아 반복이 끝나지 않는다.
- decode — 바이트 오프셋 at 에서 코드포인트 하나를 읽어 **값**을 낸다. at 이 끝을 넘음, 선두가 이어지는 바이트, 시퀀스가 슬라이스 밖으로 잘림, 이어지는 바이트가 10xxxxxx 가 아님, 과길이, 서로게이트, U+10FFFF 초과면 none 이다. 한 글자가 최대 4 바이트에 걸치므로 슬라이스 전체를 받고, len s 가 잘림의 판정 근거다. at 은 반드시 글자의 **시작**이어야 한다 — next 가 준 위치만 넣는 것이 안전한 습관이다. 길이는 주지 않는다.
- next — 다음 코드포인트의 **시작 위치**. 무효면 none 이다 — 건너뛰며 뭉개지 않는다. 선두·길이·잘림만 보고 값 범위 검사는 decode 의 몫이다. 돌려받은 값을 그대로 다음 at 으로 쓰는 것이 순회 한 걸음이다.
- count_chars — 전체를 훑어 코드포인트 수를 낸다. 무효 바이트를 만나면 none. “전체” 가 계약이라 시작 위치를 받지 않는다. 비용은 O(바이트 수)다.
- is_valid — 전체가 올바른 UTF-8 인가. “검증은 따로 된 op 이지 타입 불변식이 아니다” 의 실물이고, 구현은 is_some (count_chars s) 다. 글자 수까지 필요하면 count_chars 를 한 번만 부르는 편이 낫다.

쓰는 법

먼저 검증하고, decode 로 값을 읽고, next 로 전진한다. "한" 은 바이트 3, 글자 1, 값 54620(U+D55C)이다.

```
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
```

출력은 `U+61` · `U+d55c` · `U+1f600` 세 줄이다. 한 바퀴에서 `i` 는 `a`(1 바이트) → `한`(3 바이트) → `😀`(4 바이트)를 지나며 0 → 1 → 4 → 8 로 뛴다. **`add i 1` 로는 전진하지 않는다** — 둘째 바퀴에서 글자 한가운데를 가리키게 된다. 글자 수만 필요하면 `let n option u64 . be utf8.count_chars s .` 한 줄이다.

반례

> **반례. `none` 을 치환 문자로 때우고 계속 간다**
>
> `next` 가 실패했을 때 `set i (add i 1)` 로 한 바이트 건너뛰면 무효 입력이 조용히 통과하고, 그 위의 비교와 필터가 거짓 전제 위에 선다. 오류는 없다 — **조용히 틀린 답**이고 글자 수만 슬쩍 는다. 무효 입력을 일부러 넣어 `is_valid` 가 `false` 인데도 뒤쪽 처리가 도는지 본다. 고치는 법은 `none` 에서 멈추는 것뿐이다.

> **반례. 바이트 오프셋을 글자 번호로 여긴다**
>
> `decode s 1` 은 “둘째 글자” 가 아니라 “바이트 1 에서 읽기” 다. `"한글"` 의 바이트 1 은 시퀀스 한가운데라 `none` 이다. `is_valid` 는 `true` 인데 특정 위치만 `none` 이면 그 위치가 글자 경계가 아니다.

> **반례. `count_chars` 의 `none` 을 0 으로 다룬다**
>
> `none` 은 "글자 없음" 이 아니라 "무효 UTF-8" 이다. 빈 슬라이스는 `some 0` 이다. 둘을 합치면 깨진 입력과 글자가 없는 입력을 영영 구별할 수 없다. 검사 없이 `some_value` 로 꺼내면 실행 중 `E-VM-NONE` 으로 멈춘다.

> **반례. 검증하지 않은 입력을 `subslice` 로 자른 뒤 올바르다고 여긴다**
>
> 바이트 자르기는 늘 성공하지만 시퀀스 한가운데를 자를 수 있다. 원본은 멀쩡한데 조각에서만 `is_valid` 가 `false` 면 자른 자리가 글자 한가운데였다. 자르는 그 자리에서는 아무도 경고하지 않고, 한참 뒤 출력이 깨져야 드러난다. 글자 경계로 자르려면 `next` 가 준 위치에서만 자른다.

주의

- **`len s` 는 바이트 수다.** 글자 수는 `count_chars` 만이 안다. "최대 10 글자" 를 `len` 으로 재면 어긋난다(`"한글"` 은 `len` 6, 글자 2).
- **커서는 `next` 로 민다.** 1 씩 밀면 이어지는 바이트를 선두로 읽어 `none` 이 나오거나 글자 수가 부푼다.
- **`decode` 와 `next` 의 `none` 은 판정 범위가 다르다.** `next` 는 구조만 보고 값 범위(과길이·서로게이트·상한)는 보지 않는다. 엄밀한 순회는 둘을 함께 쓰거나 먼저 `is_valid` 로 전체를 검증한다.
- **`count_chars`·`is_valid` 는 O(바이트 수)다.** 반복 조건 안에서 매번 부르면 전체가 $O(n^2)$ 가 된다 — 한 번 세어 들고 다닌다.
- **코드포인트 단위다, 사람이 세는 글자 단위가 아니다.** 조합 문자와 이모지 시퀀스는 여러 코드포인트로 센다(국기 이모지 하나가 코드포인트 둘).
- **모두 `effects none` 이다.** `fn`·계약·`comptime` 어디서든 부를 수 있다.

utf16 — UTF-16 서로게이트 산술

소스 lib/utf16.low
층 L0 — 순수 계산
권한 없음

UTF-16 의 서로게이트 쌍을 코드포인트로 바꾸고 되돌린다. Windows API·Java·JavaScript 처럼 문자열을 16 비트 칸(unit)으로 드는 세계와 값을 주고받을 때 쓴다. `slice u16` 이 곧 UTF-16 문자열이고, 한 원소가 한 칸이다.

```
use utf16 .

let c option u64 . be utf16.decode s 0 .
guard is_some c . else return 1 .
```

`0` 은 "첫 글자" 가 아니라 **0 번 칸**이다. 이 모듈의 위치는 모두 칸 번호이고, 글자 단위 전진은 `next` 가 한다(짝이면 두 칸을 뛴다).

왜 서로게이트가 생겼는가. UTF-16 은 처음에 "문자 하나 = 16 비트 한 칸" 으로 모든 글자를 담으려 했다. 유니코드가 65,536 자를 넘어 자라자 값 구간 U+D800 … U+DFFF 를 문자로 쓰지 않기로 비워 두고, 그 구간의 두 칸을 조합해 큰 코드포인트를 적기로 했다. 상위 1024 가지 × 하위 1024 가지 = 1,048,576 가지로 U+10000 … U+10FFFF 를 딱 덮는다. 그래서 UTF-16 에서는 **칸 하나를 읽는 것과 문자 하나를 읽는 것이 다르다**.

왜 라이브러리인가. `slice u16`·`slice u32` 가 읽기·쓰기·색인을 이미 모두 한다. 권한도 필요 없고 언어에 없는 표현도 필요 없다. 그래서 새 타입도 새 낱말도 내장 연산도 없이 라이브러리가 되었다. 남는 진짜 일은 **서로게이트 산술**과 그 실패 방식뿐이다. UTF-32 라이브러리는 없다 — `slice u32` 의 한 칸이 곧 코드포인트라 필요한 것은 검증 한 줄(`cp_valid`)이다.

설계와 경계

- **하는 것** — 코드포인트와 UTF-16 칸 사이의 변환, 그리고 틀린 것의 거절. 짝 없는 상위 서로게이트(D800 … DBFF 뒤에 하위가 오지 않음), 짝 없는 하위 서로게이트(DC00 … DFFF 가 혼자 옴), 범위 밖 코드포인트(> U+10FFFF)는 `none` 이다.
- **`utf8` 과 짝이 맞는다.** `utf8` 이 UTF-8 안의 서로게이트를 거절하므로, 서로게이트는 **오직 UTF-16 안에서, 오직 짝으로만** 나타난다. 두 라이브러리가 함께 그 불변을 지킨다.
- **하지 않는 것** — 정규화(NFC·NFD), 대소문자, 조합 문자, 바이트 순서(BOM·빅/리틀 엔디언). `slice u16` 은 이미 기계 차례다.
- **모두 `effects none` 이다.** `put` 은 `proc` 이지만 자기 상태가 아니라 호출자가 건넨 버퍼에만 쓰므로 역시 `effects none` 이다.

BMP(U+0000 … U+FFFF, 서로게이트 구간 제외)의 코드포인트는 한 칸, 값 그대로다. U+10000 … U+10FFFF 는 두 칸이다 — `v = c - 0x10000` 으로 20 비트를 만들고 상위 칸 `hi = 0xD800 + v/1024`, 하위 칸 `lo = 0xDC00 + v%1024` 로 가른다. 되돌리는 산술은 `c = 0x10000 + (hi - 0xD800) × 1024 + (lo - 0xDC00)` 이다.

칸 값(10 진)	정체	뜻
55296 … 56319	상위 서로게이트	두 칸짜리 문자의 **앞** 조각 — 뒤에 하위가 와야 한다
56320 … 57343	하위 서로게이트	두 칸짜리 문자의 **뒤** 조각 — 혼자 오면 틀렸다
그 밖	평범한 BMP 칸	그 값이 곧 코드포인트다

표 50.1 — 칸 하나의 정체

코드포인트 상한은 1114111(U+10FFFF)이다. 서로게이트 값 자체는 코드포인트가 아니고 `cp_valid` 가 거절한다.

op 한눈에

op	시그니처	안 될 때
`is_high`	`fn (u u64) → bool`	실패 없음
`is_low`	`fn (u u64) → bool`	실패 없음
`cp_valid`	`fn (c u64) → bool`	실패 없음(거짓이 곧 답)
`units`	`fn (c u64) → u64`	실패 없음(1 또는 2)
`unit_hi`	`fn (c u64) → option u64`	유효하지 않은 코드포인트면 `none`
`unit_lo`	`fn (c u64) → option u64`	무효 코드포인트나 BMP(칸 하나)면 `none`
`put`	`proc (dst mut slice u16, at u64, c u64) → option u64`	무효 코드포인트나 자리 부족이면 `none`
`decode`	`fn (s slice u16, at u64) → option u64`	범위 밖이거나 짝이 맞지 않으면 `none`
`next`	`fn (s slice u16, at u64) → option u64`	끝이거나 망가진 자리면 `none`
`count_chars`	`fn (s slice u16) → option u64`	망가진 자리를 만나면 `none`
`is_valid`	`fn (s slice u16) → bool`	실패 없음

표 50.2 — `utf16` 의 op

실무에서 가장 많이 쓰는 것은 `put`(쓰기)과 `decode` + `next`(순회)다.

op 상세

- **`is_high` · `is_low`** — 칸 하나의 값이 상위(D800 … DBFF)·하위(DC00 … DFFF) 서로게이트인가. 값 범위만으로 끝나는 판정이다.
- **`cp_valid`** — 코드포인트로서 유효한가. `c > 1114111` 이거나 서로게이트 구간(55296 … 57343)이면 `false`. 바깥에서 들어온 수를 쓰기 전에 한 번 거르는 문이다.
- **`units`** — 코드포인트 하나가 먹는 칸 수. `c ≥ 65536` 이면 2, 아니면 1. 버퍼 크기를 미리 셈할 때 쓴다. **유효성은 보지 않는다.**
- **`unit_hi`** — 상위 칸. BMP 면 `some c`, 두 칸이면 `some (0xD800 + v/1024)`, 유효하지 않으면 `none`.
- **`unit_lo`** — 하위 칸. BMP 면 `none` — 여기서는 오류가 아니라 "칸이 하나뿐" 이라는 뜻이다. 두 칸이면 `some (0xDC00 + v%1024)`. 무효 코드포인트도 `none`.
- **`put`** — 코드포인트를 `dst` 의 `at` 부터 써 넣고 **다음에 쓸 위치**(`some (at+1)` 또는 `some (at+2)`)를 돌려준다. `decode`·`next` 와 같은 위치 축이다. 몇 칸 썼는지가 필요하면 `units c` 가 답한다. 유효하지 않거나 `at + units c > len dst` 면 `none` 이고 **한 칸도 쓰지 않는다**. 스스로 메모리를 만들지 않으므로 쓸 자리를 밖에서 받는다.
- **`decode`** — `s` 의 `at` 에서 코드포인트 하나를 읽는다. 평범한 칸이면 그 값, 상위 서로게이트면 바로 다음 칸이 하위인지 확인하고 합친다. `at ≥ len s`, 하위가 혼자 옴, 상위 뒤에 하위가 오지 않음, 상위로 끝남(잘림)이면 `none`. 짝이 두 칸이라 슬라이스 전체를 받는다. `at` 은 문자의 **시작** 칸이어야 한다.
- **`next`** — 다음 코드포인트의 자리. `decode` 가 성공해야 하고, 상위 서로게이트에서 시작했으면 `some (at+2)`, 아니면 `some (at+1)`. 짝을 두 칸으로 건너뛰는 셈을 호출자가 하지 않게 해 준다.

- `count_chars` — 전체를 훑어 코드포인트 수를 센다. 망가진 자리를 만나면 **수를 내지 않는다** — 틀린 수를 내느니 내지 않는다. O(칸 수).
- `is_valid` — `count_chars` 가 `some` 인가. 짝의 반쪽만 떼어 보면 무효로 보이므로, 유효성은 부분으로 나눠 묻지 않는다.

쓰는 법

쓰기는 `put` 을 부르고 돌려받은 다음 위치에서 이어 쓴다. 읽기는 `decode` 로 값을 읽고 `next` 로 전진한다. 어느 쪽이든 커서는 호출자가 든다.

```
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 라서 한 칸. put 은 다음에 쓸 위치를 돌려준다
  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 --- BMP 밖이라 두 칸(서로게이트 짝)으로 갈라진다
  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 다시 하나로 읽는다 --- decode 가 짝을 합친다
  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' + 이모지(짝) + '가' = 칸 넷, 문자 셋
  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
```

상위 칸 `buf[1]` 은 55357(0xD83D), 하위 칸 `buf[2]` 는 56489(0xDCA9)다. 순회는 `next` 로 한다.

```
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
```

BMP 면 +1, 짝이면 +2 — 직접 세지 않는다. `none` 이면 멈춘다(끝인지 망가짐인지는 먼저 `is_valid` 를 재면 갈린다).

반례

반례. 짝이 맞지 않는 입력을 `decode` 로 통과시키려 한다

```
set (index buf 0) 55357 .          rem ✗ 상위 0xD83D 뒤에
set (index buf 1) 65 .             rem   하위가 아니라 'A' 가 온다
let a option u64 . be u.decode buf 0 .
```

`none` 이다. 하위 서로게이트가 혼자 오거나(`decode buf 2`, 값 56489), 상위 서로게이트로 끝나도(`decode (subslice buf 0 4) 3`) 마찬가지다. 셋 다 `guard is_some` 자리에서 바로 걸린다. 조용히 통과시키면 그 뒤가 모두 틀리므로 **거절이 곧 이 라이브러리의 값**이다. 온전한 짝은 여전히 통과한다.

반례. 두 칸짜리를 마지막 한 칸에 넣으려 한다

`u.put buf (sub (len buf) 1) 128169` 는 `none` 이고 **그 칸은 더럽혀지지 않는다**. 반쯤 쓰다 말았나 걱정하지 않아도 된다.

반례. 서로게이트 값을 코드포인트로 쓰려 한다

`u.cp_valid 55357` 은 `false` 이고, 그래서 `put·unit_hi·unit_lo` 가 모두 `none` 이다. 이 `none` 들을 검사 없이 `some_value` 로 꺼내면 실행 중 `E-VM-NONE` 으로 멈춘다 — 번역은 통과하므로 `guard` 가 먼저다.

주의

- **`len s` 는 칸 수다.** 문자 수는 `count_chars` 만이 안다. 이모지 하나가 칸 둘을 먹는다.
- **짝 한가운데서 자르지 않는다.** 상위와 하위 사이를 `subslice` 로 가르면 양쪽 조각이 모두 무효가 된다. 자를 자리는 `next` 가 준 값이어야 한다.
- **커서는 `next` 로 민다.** 1 씩 밀면 짝의 하위 칸을 문자 시작으로 읽어 `none` 이 나오거나 문자 수가 부푼다.
- **비용.** `decode·next·put` 은 O(1), `count_chars·is_valid` 는 O(칸 수)다. 반복 조건 안에서 매번 세지 않는다.
- **`unit_lo` 의 `none` 은 두 뜻이 겹친다**(무효 코드포인트 · BMP). 구분이 필요하면 `cp_valid` 를 먼저 묻는다. 조합해 쓸 때는 실패가 `none` 하나로 정리되는 `put` 이 낫다.
- **바이트 직렬화는 밖이다.** 파일이나 네트워크에서 온 **바이트**를 `slice u16` 으로 만드는 일은 호출자 몫이다.

unicode — 유니코드 속성 표

소스 lib/unicode.low
층 L0 — 순수 계산
권한 없음

"이 글자가 문자인가, 숫자인가, 공백인가" 를 묻는 곳이다. ASCII 만 다룰 때는 `ge c 97` 이고 `le c 122` 면 소문자라고 손으로 답할 수 있었다. `'한'` 이나 `'あ'` 나 `'Ω'` 가 들어오는 순간 그 방식은 무너진다. 유니코드에서 "문자" 인 코드포인트는 **658 개의 흩어진 구간**에 걸쳐 있어 손으로 적지 못한다. 이 모듈은 그 구간표를 **Unicode 15.1.0 원본에서 기계적으로 뽑아** 싣고 이진 탐색으로 답한다.

```
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')`은 `true`, `is_letter 128512(😀)`는 `false` 다 — 이모지는 기호이지 문자가 아니다. **입력은 코드포인트(`u64`)이지 바이트가 아니다.** UTF-8 바이트열에서 코드포인트를 꺼내는 일은 `utf8` 이 하고, 둘을 이어 쓴다.

표는 데이터다 — 코드로 쓰면 거짓말이 섞인다. 이 모듈이 없으면 사람마다 자기 프로그램에 글자 판정을 손으로 적고, 그 판정은 거의 반드시 틀린다. 문제는 틀리는 방식이다. 빠뜨린 구간은 오류가 아니라 `false` — **조용히 틀린 답**으로 나온다. 파서라면 그 글자에서 토큰이 끊기고, 검색이라면 그 낱말이 안 잡히며, 어디에서도 경고가 뜨지 않는다. 표시 폭(`term` 의 `cp_width`)은 틀려도 1 이라는 쓸 만한 기본값이 있었지만, 여기에는 그런 기본값이 없다.

설계와 경계

- **표를 고정폭 16 진 문자열 하나로 싣는다.** 구간 하나가 `<시작 6 자리><끝 6 자리>` = 12 바이트다. `n` 번째 구간의 위치가 `n × 12` 로 바로 나오므로 해독 없이 이진 탐색이 된다. 펼칠 배열도, 초기화 코드도, 할당도 없어 층 전체가 `effects none` 으로 남고 VM·네이티브 대조 안에서 검증된다.
- **범위는 U+0000 … U+2FFFF 다.** 그 위는 아직 거의 쓰이지 않는다. **담지 않은 것을 담은 척하지 않는다** — U+30000 이상은 모두 "속성 없음" 으로 답한다.
- **세부 범주는 물을 곳이 생길 때 나눈다.** 정규식 `\p{Lu}` 가 물을 곳을 만들었을 때 Lu·Ll·Nd 를 나눴고 이어서 나머지도 나눴다. 이제 L 과 N 의 하위가 모두 있다(L = Lu+Ll+Lt+Lm+Lo, N = Nd+Nl+No). P·Z·M 의 하위는 아직 나누지 않았다.
- **표는 다시 뽑을 수 있다.** 개발 저장소의 추출 스크립트가 표를 뽑고, **실린 표가 추출과 바이트까지 같은지** 매번 검사한다. 손으로 한 번 붙여 넣은 표는 그 순간부터 검증할 수 없는 상수 덩어리다. 이 검사가 실제로 잡은 것도 있다 — 옛 문서가 `is_space` 에 탭과 개행이 든다고 적었는데 표는 그렇지 않았다.
- **변환은 구간이 아니라 짝이다.** 범주는 "이 구간에 드는가" 를, 변환은 "**어디로** 가는가" 를 묻는다. 그래서 `tab_toupper·tab_tolower` 는 같은 고정폭이되 `<from><to>` 짝을 싣는다. **1 대 1 만** 싣고 그 한계를 숨기지 않는다(`is_special_case`).
- **짓지 않은 것** — 정규화(NFC·NFD), 조합, P·Z·M 의 하위 범주, 문자 이름, 전체 대소문자 폴딩(ß → ss 처럼 길이가 바뀌는 변환). 모두 또 다른 표가 필요하고 표는 공짜가 아니다.

표는 `fn` 이 돌려주는 `slice u8`(문자열 리터럴 — 복사도 할당도 없다)이고, 구조체도 상태도 없다. 표의 불변식은 넷이다 — 구간은 소문자 16 진 12 바이트, 시작 오름차순 정렬, 겹침 없음(인접 구간은 병합), **끝 포함**(`start ≤ cp ≤ end`). 표를 직접 만들어 `has_cp` 에 넘긴다면 정렬과 겹침 없음을 지켜야 이진 탐색이 선다.

op 한눈에

op	하는 일
`tab_letter` · `tab_number`	문자(L) 표 658 구간 · 숫자(N) 표 137 구간
`tab_punct` · `tab_space` · `tab_mark`	문장부호(P) · 공백(Z, 탭과 개행은 들지 않는다) · 결합 표시(M) 표
`tab_zerowidth`	**폭 0**(Mn·Me·Cf) 표 354 구간
`tab_upper` · `tab_lower` · `tab_digit`	대문자(Lu) 646 · 소문자(Ll) 658 · 십진 숫자(Nd) 64 구간
`tab_title` · `tab_modifier` · `tab_other_letter`	Lt 10 · Lm 71 · Lo 509 구간
`tab_numletter` · `tab_numother`	Nl 12 · No 72 구간
`tab_toupper` · `tab_tolower`	대소문자 **짝** 표 1423 · 1432 짝
`tab_casespecial`	1 대 1 로 바꾸지 못하는 코드포인트 103 개
`range_count`	표의 구간 개수(`len tab / 12`)
`has_cp`	표에 이 코드포인트가 드는가 — 이진 탐색
`is_letter` · `is_number` · `is_punct` · `is_space` · `is_mark`	각 표의 술어
`is_alnum`	문자 또는 숫자
`is_upper` · `is_lower` · `is_digit`	Lu · Ll · **십진** 숫자(Nd)인가
`is_zerowidth`	폭 0(결합 문자·서식 문자)인가
`is_title` · `is_modifier` · `is_other_letter` · `is_numletter` · `is_numother`	Lt · Lm · Lo · Nl · No 인가
`to_upper` · `to_lower`	대문자로 · 소문자로(바뀌지 않으면 **그대로**)
`is_special_case`	1 대 1 로 바꾸지 못하는 코드포인트(ß 따위)인가

표 50.1 — `unicode` 의 op

op 상세

- **`tab_*`** — 매개변수가 없고, 프로그램 수명 내내 유효한 읽기 전용 뷰를 돌려준다. 직접 쓸 일은 여러 표를 한 반복에서 돌리거나 `range_count` 로 크기를 볼 때 정도다.
- **`range_count tab`** — `len tab / 12`. 표가 아닌 슬라이스를 넘기면 뜻 없는 수가 나온다(검사하지 않는다).
- **`has_cp tab cp`** — 핵심. 비용은 O(log n) 비교이고 비교마다 6 자리 16 진을 두 번 읽는다. 할당도 버퍼도 없다. **표가 망가졌으면**(16 진이 아닌 바이트) `false` 를 돌려준다 — 지어내지 않는다는 뜻이자 망가진 표를 조용히 넘어간다는 뜻이다.
- **`is_letter` 따위** — `has_cp <해당 표> cp` 의 얇은 껍질이다. 어떤 `u64` 를 넣어도 답한다(범위 밖이면 `false`). `is_alnum` 은 식별자 검사에 가장 자주 쓰는 조합이라 따로 뒀다.

- **`is_digit` 는 `is_number` 보다 좁다.** 로마 숫자 Ⅷ(U+2166)나 분수 ½ 는 N 이지만 Nd 가 아니다. 자릿수를 계산하려면(`d = cp - '0'`) 반드시 Nd 여야 한다.
- **한글은 `is_upper` 도 `is_lower` 도 `false` 다**(Lo — 대소문자가 없는 문자). 그래도 `is_letter` 는 `true` 다. "대문자가 아니다" 가 "문자가 아니다" 는 아니다.
- **`is_zerowidth`** — Mn(결합 표시)·Me(둘러싸는 표시)·Cf(서식 문자)에 `true`. 이 코드포인트들은 자기 칸을 차지하지 않고 앞 글자에 얹힌다. `term` 의 폭 계산과 글자 묶음 세기가 **같은 표**에서 나오므로 "폭은 맞는데 커서 이동은 틀리는" 어긋남이 생기지 않는다. `is_mark` 는 M 범주 전체(자기 칸을 차지하는 Mc 포함)라서, 폭을 세려면 `is_zerowidth` 를 쓴다.
- **`to_upper`·`to_lower`** — 바뀌지 않으면 그대로 돌려준다(`none` 이 아니다). 세상 코드포인트의 대부분은 대소문자가 없고, 그때마다 `option` 을 풀게 하면 호출부가 온통 `some_value` 가 된다. **1 대 1 만 한다** — 독일어 ß(U+00DF)의 대문자는 두 글자 `"SS"` 라 짝으로 못 적으므로 `to_upper 223` 은 223 을 그대로 준다. 그런 코드포인트는 `is_special_case` 가 따로 말한다. 왕복도 늘 성립하지 않는다(터키어 `ı`, 그리스어 종결 시그마). 대소문자 무시 비교는 폴딩이고 아직 없다.

쓰는 법

UTF-8 문자열에서 낱말(문자·숫자가 이어진 덩어리)을 센다.

```
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"` 에서 3 이 나온다 — ASCII 만 아는 판정으로는 `'한글'` 을 세지 못한다. 정규식의 `\p{L}`·`\P{L}` 이 바로 이 표를 쓴다 (`regex`). 정규식으로 충분하면 정규식을 쓰고, 코드포인트 하나만 물으면 되는 자리에서는 이 모듈을 직접 부르는 편이 훨씬 싸다.

반례

반례. 바이트를 그대로 넘긴다

```
rem ✘ index 는 바이트를 준다. '한' 의 첫 바이트는 0xED 다
if unicode.is_letter (widen u64 (index s 0)) . do … end
```

0xED(237)는 코드포인트 U+00ED(í)다 — 우연히 문자로 판정되지만 **묻고 있던 글자가 아니다**. UTF-8 을 다루면 반드시 `utf8.decode` 를 거친다.

반례. 폭을 세면서 `is_mark` 를 쓴다

Mc(자기 칸을 차지하는 결합 표시)까지 0 으로 세게 된다. 폭에는 `is_zerowidth` 다 — 애초에 폭은 `term` 의 `cp_width` 가 다 처리한다.

반례. `to_upper` 가 늘 바꿔 준다고 여긴다

ß 는 그대로 온다. 정확해야 하면 먼저 `is_special_case` 로 묻고, 두 글자 이상으로 펼치는 일은 호출자가 정한다.

반례. `is_number` 로 자릿수를 계산한다

```
rem ✘ Ⅷ(U+2166)도 통과하고 cp - 48 은 뜻 없는 수다
if unicode.is_number cp . do set v (add (mul v 10) (sub cp 48)) . end
```

십진 숫자만 받으려면 `is_digit` 이다.

반례. `is_upper` 가 `false` 라서 소문자로 여긴다

한글·한자·아랍 문자는 둘 다 `false` 다. 그 판정은 세상 대부분의 글자를 소문자로 분류한다. 소문자인지는 `is_lower` 로 직접 묻는다.

주의

- **범위 밖은 조용히 `false` 다.** U+30000 이상을 다루는 프로그램이라면 이 모듈이 답하지 못한다는 것을 알고 쓴다.
- **`has_cp` 의 `false` 는 "들지 않는다" 와 "표가 깨졌다" 를 구별하지 않는다.** 내장 표는 늘 온전하다. 표를 직접 만든다면 넘기기 전에 `mod (len tab) 12` 가 0 인지 확인한다.
- **표는 Unicode 15.1.0 기준이다.** 유니코드가 갱신되면 표를 손으로 고치지 않고 다시 뽑는다.
- **바이너리 크기.** 표 전체가 문자열로 들어간다(수십 KB). 작은 기계에서 문제가 되면 필요한 표만 쓰는 모듈을 따로 두는 것이 맞다.
- **`is_space` 는 Z 범주 그대로다 — 탭·개행·CR 은 들지 않는다.** 유니코드에서 그것들은 제어 문자(Cc)다. 정규식의 `\s` 처럼 굴기를 바란다면 호출부에서 더한다. 이 표가 `\p{Z}` 에 그대로 쓰이기 때문에 섞지 않았다.

codec — 16 진과 base64

소스 lib/codec.low
층 L0 — 순수 계산(호출자의 버퍼)
권한 없음

바이트를 16 진이나 base64 **글자열로 바꾸고 되돌린다**. 세상에는 텍스트만 통과시키는 자리가 많다 — 설정 파일 한 줄, 로그, URL, 메일 본문, 화면 출력. 그런 자리에 바이트를 그대로 넣으면 0 바이트나 제어 바이트가 끼어 통로가 망가진다. 16 진(바이트 하나를 `0-9 a-f` 두 글자로)과 base64(임의 바이트를 `A-Z a-z 0-9 + /` 64 가지 글자로)는 **임의 바이트를 안전한 글자만으로 나르는 포장지**다. 16 진은 사람이 읽기 좋지만 크기가 정확히 2 배가 되고, base64 는 읽기 어렵지만 약 4/3 배로 끝난다. 해시값을 눈으로 볼 때는 16 진, 덩어리를 실어 나를 때는 base64 가 보통이다.

```
use codec as c .

let n option u64 . be c.hex_enc "abc" dst .
guard is_some n . else return 1 .
```

`dst` 는 이 모듈이 만드는 것이 아니라 **호출자가 미리 잡아 두는 출력 자리**다(여기서는 6 바이트 이상). `n` 은 **실제로 쓴 바이트 수**이고 결과는 `dst[0..n)` 이다.

포장은 비밀을 지키지 않는다 — 누구나 되돌린다. 비밀이 필요하면 봉인하는 `aead` 를 쓴다(33장). 이 모듈은 바이트에서 바이트로 가는 순수 계산이라 `index·set·shl·shr·bit_and·bit_or·guard·while` 만 쓰고, 내장 연산은 하나도 늘지 않았다.

설계와 경계

- **잘못된 입력은 값으로 답한다 — `none` 하나다.** 홀수 길이 16 진, 16 진이 아닌 글자, base64 가 아닌 글자, 어긋난 패딩, **모자란 출력 버퍼** 모두 `none` 이다. 멈추지도 자르지도 않는다 — 잘라 주면 뒤의 모든 층이 "다 썼다" 는 거짓 전제 위에 선다. 성공은 `some <쓴 바이트 수>` 이고, 호출자가 `subslice` 로 정확한 결과를 얻는다.
- **16 진 인코딩은 소문자를 낸다.** 해독은 대소문자를 둘 다 받는다 — 관대하게 받고 보수적으로 보낸다.
- **base64 는 표준 알파벳**(`A–Z a–z 0–9 + /`)과 `=` 패딩이다. base64 는 입력을 3 바이트씩 묶어 4 글자로 적고, 마지막 묶음이 모자라면 `=` 로 메워 길이를 늘 4 의 배수로 맞춘다.
- **짓지 않은 것** — URL 에 안전한 변형(`-_`), 개행 접기. 필요해지면 **따로 된 op** 으로 짓는다 — 한 op 에 모드 인자를 다는 것이 곧 혼란이다.
- **출력 버퍼는 호출자의 것이다.** 버퍼를 마련하는 비용과 정책은 호출자만 안다. 그래서 `effects none` 이고 숨은 할당이 없다.

op	필요한 `dst` 크기	성공 반환
`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) − 패딩 수`	`some <쓴 바이트 수>`

표 50.1 — 필요한 출력 자리

`dst` 가 공식보다 작으면 결과는 늘 `none` 이고, 크면 상관없다 — 앞에서부터 쓰고 쓴 수를 돌려준다. 빈 입력은 성공이다(`hex_enc ""·b64_enc ""·b64_dec ""` 모두 `some 0`).

op 한눈에

op	시그니처	안 될 때
`hex_digit`	`fn (v u64) → u8`	실패 없음(0 … 15 가 전제)
`hex_val`	`fn (c u8) → option u64`	`0-9 a-f A-F` 밖이면 `none`
`hex_enc`	`proc (src slice u8, dst mut slice u8) → option u64`	`dst` 부족이면 `none`
`hex_dec`	`proc (src slice u8, dst mut slice u8) → option u64`	홀수 길이 · 16 진이 아닌 글자 · `dst` 부족이면 `none`
`b64_digit`	`fn (v u64) → u8`	실패 없음(0 … 63 이 전제)
`b64_val`	`fn (c u8) → option u64`	알파벳 밖이면 `none`(`=` 도 `none`)
`b64_enc`	`proc (src slice u8, dst mut slice u8) → option u64`	`dst` 부족이면 `none`
`b64_dec`	`proc (src slice u8, dst mut slice u8) → option u64`	길이가 4 의 배수가 아님 · 알파벳 밖 · 어긋난 패딩 · `dst` 부족이면 `none`

표 50.2 — `codec` 의 op

보통은 `hex_enc`·`hex_dec`·`b64_enc`·`b64_dec` 넷만 쓰고, 나머지 넷은 글자 하나짜리 부품이다.

op 상세

모든 변환이 `src`(읽을 바이트)와 `dst`(쓸 자리)를 따로 받는다. 제자리 변환을 하지 않는 것은 인코딩이 결과를 키워서 원본을 덮으면 아직 읽지 않은 바이트를 밟기 때문이다. **`src` 의 길이가 곧 입력의 전부**다 — 끝 표식을 찾지 않고 `len src` 만 믿으므로, 큰 버퍼에 든 결과를 다시 넘길 때는 `subslice` 로 정확히 잘라 준다. 결과의 길이는 `dst` 의 길이가 아니라 반환된 `some n` 의 `n` 이다.

- **`hex_digit v`** — 니블(0 … 15)을 소문자 16 진 글자로. `v < 10` 이면 `'0'+v`, 아니면 `'a'+(v−10)`. 범위를 검사하지 않는 대신 호출자가 보장한다.
- **`hex_val c`** — 16 진 글자 하나의 값. 대소문자를 둘 다 받고, 밖이면 `none`.
- **`hex_enc src dst`** — 먼저 `2×len src ≤ len dst` 를 검사하고, 모자라면 아무것도 쓰지 않고 `none`.
- **`hex_dec src dst`** — 길이가 짝수이고, 모든 글자가 16 진이고, `len src / 2 ≤ len dst` 여야 한다. 16 진이 아닌 글자는 **훑다가** 발견되므로 그 앞까지의 바이트는 이미 `dst` 에 쓰였을 수 있다. `none` 이면 `dst` 의 내용은 정해지지 않은 것으로 다룬다.
- **`b64_digit v`** — 표준 알파벳의 `v`(0 … 63)번째 글자. 0 … 25 는 `A-Z`, 26 … 51 은 `a-z`, 52 … 61 은 `0-9`, 62 는 `+`, 63 은 `/`. base64 는 바이트가 아니라 6 비트 조각을 글자에 대응시킨다.
- **`b64_val c`** — base64 글자 하나의 값. `=` **도 `none`** 이다 — 패딩은 글자가 아니라 해독기 본문이 다루는 구조다.
- **`b64_enc src dst`** — `=` 패딩을 포함해 쓴다. 꼬리가 1 바이트면 `XX==`, 2 바이트면 `XXX=`.
- **`b64_dec src dst`** — 길이가 4 의 배수이고, 모든 글자가 알파벳이거나 규칙에 맞는 패딩이고, `dst` 가 충분해야 한다. `=` 는 **마지막 묶음**에만 올 수 있고, 셋째 자리가 `=` 면 넷째도 `=` 여야 한다. 성공하면 패딩만큼 줄어든 정확한 길이를 준다. `none` 이면 `dst` 의 내용은 정해지지 않았다.

쓰는 법

버퍼 크기를 먼저 확인하고, 변환하고, 돌려받은 개수로 잘라 쓴다.

```
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 쓴 수가 곧 경계다 --- enc 통째로 넘기면 뒤의 쓰레기까지 입력이 된다
  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 대문자도 받는다: "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 바이트는 한 묶음(4 글자)이 되고 끝에 = 하나가 붙는다: "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
```

반례

모두 멈춤이 아니라 `none` 이다. 증상은 늘 "결과가 `none` 이고 `guard` 가 `else` 로 빠지는 것" 이다.

입력	결과	흔한 원인과 `dst`
`c.hex_dec "abc" dst`	none	홀수 길이 — 길이 검사에서 먼저 걸려 `dst` 는 그대로
`c.hex_dec "zz" dst`	none	16 진이 아닌 글자 — 훑다가 발견되므로 `dst` 앞부분이 덮였을 수 있다
`c.hex_enc "abcdefgh" dst3`	none	3 바이트 자리에 16 이 필요 — 한 바이트도 쓰지 않는다
`c.b64_dec "abc" dst`	none	길이가 4 의 배수가 아님 — `enc` 를 `subslice` 로 자르지 않고 통째로 넘긴 경우가 흔하다
`c.b64_dec "a?cd" dst`	none	알파벳 밖 — URL 에 안전한 `-·_` 나 개행이 섞인 경우가 대부분
`c.b64_dec "aG==YWJj" dst`	none	가운데 패딩 — 두 base64 문자열을 그냥 이어 붙이면 이 꼴이 된다

표 50.3 — 자주 틀리는 입력

주의

- **`none` 만 계속 나오면 먼저 버퍼 크기를 의심한다.** "코드는 맞는데 늘 `none`" 의 원인은 거의 늘 `dst` 를 공식보다 작게 잡은 것이다. 넉넉히 잡는 것은 손해가 없다.
- **`none` 이면 `dst` 를 결과로 쓰지 않는다.** 인코더는 크기를 먼저 검사해 실패하면 `dst` 가 그대로지만, 해독기는 훑으면서 쓰므로 앞부분이 덮였을 수 있다. 성공한 `some n` 의 `dst[0..n)` 만 결과다.
- **돌려받은 수를 버리고 `dst` 통째를 결과로 쓰지 않는다.** 뒤에 남은 옛 바이트까지 섞인다.
- **인코딩과 해독의 버퍼를 겹치지 않는다.** `src` 와 `dst` 가 겹치면 아직 읽지 않은 입력을 출력이 덮는다. 겹침은 검사하지 않고 결과가 조용히 틀린다.
- **패딩을 손으로 다루지 않는다.** `b64_val` 에 `=` 를 물으면 `none` 이다. `b64_dec` 에 맡긴다.
- **URL 에 안전한 base64 와 여러 줄 base64 는 받지 않는다** — 지원하지 않는 것이지 결함이 아니다. 넘기기 전에 호출자가 개행을 걷어 낸다.
- **선형 비용, 공짜 재진입.** 모두 입력 길이에 선형이고 상태가 없어서 여러 곳에서 동시에 불러도 간섭하지 않는다.

regex — 역추적하지 않는 Pike VM 정규식

소스 lib/regex.low
층 L0 — 순수 계산(호출자의 스크래치)
권한 없음

문자열이 **어떤 꼴인지** 검사하고 찾는다. 입력 검증(숫자만 왔는가), 로그에서 패턴 찾기, 간단한 토큰 나누기에 쓴다. 패턴을 한 번 **컴파일**해 프로그램으로 만들어 두고, 같은 프로그램으로 여러 입력을 맞춘다.

```
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 .
```

프로그램(slice u64)도, 컴파일러 상태도, 매처의 작업 공간도 모두 호출자의 슬라이스다. 그래서 effects none 이고 숨은 할당이 없으며 재진입이 공짜이고 VM· 네이티브 대조 안에서 돈다.

왜 Pike VM 이 안전한가. 흔한 정규식 엔진은 **역추적** 방식이다. 갈림길(¦·*)에서 한 길을 끝까지 가 보고 막히면 되돌아와 다른 길을 간다. 갈림길이 겹겹이 겹치면 시도할 길이 2·4·8⋯ 로 불어나고, (a*)*b 같은 패턴 하나가 입력 몇십 바이트에 몇 분씩 걸린다 — 이것이 ReDoS(정규식 서비스 거부)다. Pike VM 은 되돌아가지 않는다. 패턴을 NFA 로 보고 "지금 있을 수 있는 자리" 전부를 **스레드 목록**(OS 스레드가 아니라 프로그램의 몇 번째 명령에 서 있는지를 적은 값들)에 담아, 입력 한 바이트마다 목록 전체를 **한꺼번에 한 걸음씩** 전진시킨다. 같은 자리는 한 번만 담으므로 목록은 명령 수를 넘지 않는다. 그래서 어떤 패턴과 입력에도 **지수 폭발이 원리적으로 없다** — 시작 자리를 정한 매칭 한 번은 O(입력 × 프로그램)이다.

설계와 경계

받는 문법

- 리터럴, .(개행 제외), * + ?(탐욕), [a-z]·[^…], ¦, (…), ^ $(문자열 경계), 이스케이프 \d \w \s \n \t \r 와 특수 문자의 리터럴화(\. 따위).
- 클래스 안에서도 \n \t \r \\ \] \- 리터럴과 \d \w \s 집합의 합집합, 범위 a-z, 부정 [^…].
- {m}·{m,}·{m,n} 반복(탐욕) — **컴파일할 때 펼친다**. 실행 중 카운터를 두면 스레드마다 사적 상태가 생겨 "스레드 = 명령 위치 하나" 라는 불변이 깨지고 중복 제거가 무너진다. 상한은 64 다.
- 캡처 그룹 match_caps, UTF-8 한 글자를 먹는 \X, 유니코드 속성 \p{…}·\P{…}(L N P Z M 과 Lu Ll Lt Lm Lo Nd Nl No).

속성은 먹은 뒤에 돌아본다. 속성 집합은 구간이 수백 … 수천이라 바이트 오토마톤으로 펼치면 명령이 수천 개가 된다 — 프로그램 버퍼가 호출자의 것인 이 설계에서는 감당하지 못한다. 대신 코드포인트 하나를 먹은 **뒤에** 그 속성인지 돌아본다. UTF-8 은 자기 동기적이라 뒤로 읽을 수 있고, 그 판정은 바이트를 먹지 않으므로 선형성의 근거가 그대로다. 표는 unicode 가 든다. 모르는 속성 이름은 **거절**한다 — 넓은 표로 대신 답하면 조용히 틀린 매치가 된다.

짓지 않은 것 — 클래스 안의 속성([\p{L}0-9], 클래스는 바이트 비트맵이다), 대소문자 무시((?i), 폴딩 표가 필요하다), 역참조와 룩어라운드(이것은 짓지 않을 것에 가깝다 — 그 순간 선형성이 죽는다). 잘못된 패턴과 모자란 버퍼는 모두 **값으로**(none) 거절된다.

자료구조

프로그램 버퍼(`slice u64`) 하나에 프로그램이 통째로 산다. `prog[0]` 은 명령 수, `prog[1]` 은 클래스 수, 그 뒤로 명령이 하나에 세 워드 `[op a b]` 로 이어진다. **코드는 앞에서 자라고, 문자 클래스 비트맵(256 비트 = 4 워드)은 꼬리에서 거꾸로 자란다** — 둘이 만나면 컴파일이 `none` 이다. 클래스 k 의 비트맵은 `prog[len - 4·(k+1) .. len - 4·k)` 에 있다.

코드 · 이름	a · b	뜻
0 CHAR	바이트 값	그 바이트면 먹고 전진
1 ANY	—	개행(10) 아닌 아무 바이트를 먹는다
2 CLASS	클래스 번호 k	비트맵 k 에 있는 바이트를 먹는다
3 SPLIT	갈래 1 · 갈래 2	두 갈래를 동시에 진행(갈래 1 우선 = 탐욕)
4 JMP	목표	무조건 이동
5 MATCH	—	여기 닿으면 매치
6 BOL · 7 EOL	—	^ 은 위치 0 에서만, $ 는 `len s` 에서만 통과
8 SAVE	슬롯 번호	캡처 자리 기록(바이트를 먹지 않는다)
9 CPPROP	속성 코드	**직전에 먹은** 코드포인트가 그 속성인가(바이트를 먹지 않는다)

표 50.1 — 명령 코드

속성 코드는 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 이고, **부정은 +16** 이다(`\P{L}` = 16). 부정 자리를 넉넉히 잡아 둔 덕에 표가 8 개에서 13 개로 늘 때 인코딩이 한 비트도 바뀌지 않았다.

컴파일러 상태 `st` 는 네 칸(명령 수 · 클래스 수 · 패턴 위치 · 실패 표시)이고 `compile` 이 초기화하므로 길이 4 이상만 보장하면 된다. **매처 스크래치**는 현재·다음 스레드 목록 `clist·nlist` 와 중복 방지 도장 `marks`, 그리고 `search` 가 더 받는 시작 위치 배열 `cstart·nstart` 다. 모두 `slice u64` 이고 각각 **길이가 명령 수 이상**이어야 한다.

op 한눈에

op	시그니처	안 될 때
`compile`	`proc (pat slice u8, prog mut slice u64, st mut slice u64) → option u64`	문법 오류나 버퍼 부족이면 `none`
`match_at`	`proc (prog slice u64, s slice u8, at u64, clist, nlist, marks) → option u64`	매치 없음 · 빈 프로그램 · 스크래치 부족이면 `none`
`find`	`proc (prog slice u64, s slice u8, clist, nlist, marks) → option u64`	어디에도 매치가 없으면 `none`
`search`	`proc (prog slice u64, s slice u8, clist, nlist, marks, cstart, nstart) → option u64`	어디에도 매치 없음 · 빈 프로그램 · 스크래치 부족이면 `none`
`test_at`	`proc (prog slice u64, s slice u8, clist, nlist, marks) → bool`	실패 없음(거짓이 곧 답)
`class_has`	`fn (prog slice u64, k u64, b u64) → bool`	실패 없음

표 50.2 — `regex` 의 op — 모두 `effects none`

`rx_` 로 시작하는 이름들은 컴파일러와 매처의 내부이고 밖에서 보이지 않는다. 스크래치 매개변수는 모두 `mut slice u64` 다.

op 상세

- **`compile pat prog st`** — `len prog ≥ 8·len st ≥ 4` 여야 한다. 성공하면 `some <명령 수>` 이고, 이 값이 곧 매처 스크래치의 최소 길이다. 닫히지 않은 그룹 `(ab`, 원자 자리에 온 `)` ¦ `*` `+` `?`, 닫히지 않은 `[…]`, 뒤집힌 범위 `[z-a]`, 패턴 끝의 홀로 선 `\`, 잘못된 셈(`a{4,2}`·`a{}`·`a{2`·64 초과), 코드와 비트맵이 만남(버퍼 부족)이면 `none`. 빈 패턴은 유효하다 — `MATCH` 하나로 컴파일되어 어디서나 빈 매치다.
- **`match_at prog s at clist nlist marks`** — `at` 에서 시작하는 매치의 **가장 긴 끝**(탐욕)을 `some end`(끝은 배타적)로 준다. 시작점이 물음의 일부라 `at` 을 받는다. `clist` 는 지금 위치의 스레드 목록, `nlist` 는 다음 위치의 목록이고(한 바이트마다 둘을 맞바꾼다), `marks` 는 "이 명령은 이번 위치에 이미 담았다" 는 도장이다. 도장이 없으면 같은 자리가 여러 번 들어가 목록 길이의 상한이 깨진다. 빈 매치도 매치다(`some at`). 비용은 O(`len s` × 명령 수).
- **`find`** — 매치가 **시작되는 가장 왼쪽 위치**. `at = 0, 1, …, len s` 로 `match_at` 을 다시 시작하는 반복이라 상한은 $O(n^2 \cdot m)$ 이지만, 시도마다 선형이라 지수 폭발은 없다. 끝 위치가 필요하면 돌려받은 시작에서 `match_at` 을 한 번 더 부른다.
- **`search`** — `find` 와 같은 물음을 **한 번 훑어** 답한다. 재시작 반복을 없앤 대신 "이 스레드가 어디서 출발했는가" 를 스레드마다 들고 다녀야 해서 `cstart·nstart` 두 스크래치가 더 든다. 아직 매치가 없는 동안에만 매 위치에서 시작 스레드를 새로 넣으므로 비용은 O(입력 × 명령 수)이고, 더 왼쪽 시작이 늘 이기므로 왼쪽 우선이 지켜진다. `find` 와 **같은 답**이다 — 개발 저장소의 시험이 둘을 맞대 본다.
- **`test_at`** — 어딘가에 매치가 있는가. 위치 없이 예·아니오만 필요할 때 쓴다.
- **`class_has prog k b`** — 컴파일된 프로그램의 클래스 `k` 가 바이트 `b` 를 담는가. 비트맵이 프로그램 버퍼의 꼬리에 살아서 `prog` 를 받는다. `k` 의 범위는 검사하지 않는다.

쓰는 법

이스케이프는 두 겹이다. Lowent 문자열 리터럴의 이스케이프는 열넷으로 닫혀 있고(3장) `\d` · `\w` 같은 것은 그 안에 없다(그 밖은 `E-STR-ESCAPE`). 정규식의 `\d` 를 소스에 쓰려면 `"\\d"` 로 쓴다 — 리터럴이 `\\` 를 백슬래시 하나로 만들고, 그 `\d` 두 바이트를 정규식 컴파일러가 숫자 클래스로 읽는다.

```
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*" 를 "abcdddx" 에: . 이 c 를, 탐욕 d* 가 ddd 를 먹어 끝 = 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 정규식으로는 [a-c]+z\d --- 소스에는 "\\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$" 는 끝에서만 --- find 가 가장 왼쪽 시작을 준다
  let n3 option u64 . be rx.compile "ab$" 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 와 셈 반복도 같은 모양이다.

```
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 하한 미달은 매치가 아니다
  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
```

호출자는 예컨대 prog 32 칸, st 4 칸, cl·nl·mk(search 를 쓰면 cs·ns 까지) 각 32 칸을 마련한다 — compile 이 돌려준 명령 수보다 크면 된다. {m,n} 은 펼쳐지므로 명령 수가 반복 횟수만큼 는다.

역추적 엔진이라면 (a*)*b 에 a 64 개를 주면 2^{64} 갈래를 헤매지만, Pike VM 은 스레드 목록이라 선형으로 끝난다.

```
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
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" 0 cl nl mk .
guard eq (is_some f) false . else return 2 .
```

반례

부름	결과
`rx.compile "(ab" prog st`	`none` — 닫히지 않은 그룹
`rx.compile "*a" prog st`	`none` — 앞이 없는 반복
`rx.compile "abcdefghij" tiny st`(tiny 8 칸)	`none` — 코드와 비트맵이 만난다
`rx.match_at prog s 0 short short short`	`none` — 스크래치가 명령 수보다 짧다
`rx.compile "a{4,2}" prog st` · `"a{}"` · `"a{2"` · `"a{65}"`	`none` — 잘못된 셈(상한 64 는 펼치는 방식이라 폭발을 막는다)
`rx.compile "\d" prog st`	실행 전에 `E-STR-ESCAPE` — `"\\d"` 로 쓴다

표 50.3 — 값으로 거절되는 것과 번역에서 거절되는 것

주의

- **`prog` 는 컴파일할 때와 같은 길이로 매칭에 넘긴다.** 클래스 비트맵이 버퍼의 꼬리에 살고 그 주소가 `len prog` 기준이다. `subslice` 로 잘라 넘기면 명령은 살아도 비트맵 주소가 어긋나 `[…]`·`\d\w\s` 가 조용히 엉뚱한 바이트를 본다. 프로그램은 자르지 말고 통째로 든다.
- **프로그램 버퍼 하나에 패턴 하나.** 다시 `compile` 하면 앞 프로그램은 덮인다. 두 패턴을 번갈아 쓰려면 버퍼를 둘 마련한다.
- **`find` 냐 `search` 냐는 비용의 선택이다.** 답은 같다. 긴 입력과 잦은 탐색이면 `search`, 작은 입력에서 스크래치를 아끼려면 `find` 다.
- **`marks` 는 매번 스스로 초기화된다.** 호출 사이에 보존할 것이 없다.
- **빈 매치에 주의한다.** `a*` 같은 패턴은 어디서나 길이 0 으로 성공하고, `find` 는 늘 `some 0` 을 낸다. “한 글자라도 먹었는가” 가 필요하면 끝을 시작과 비교한다.
- **매칭 단위는 바이트다.** UTF-8 입력에서 `.`·`[^…]` 는 코드포인트가 아니라 바이트 하나를 먹는다. 코드포인트 하나는 `\X`, 유니코드 범주는 `\p{…}` 로 묻는다.

term — 터미널 렌더러(순수 절반)

소스 lib/term.low
층 L0 — 순수 계산(호출자의 버퍼)
권한 없음 — 내보내기는 outbuf 의 몫

터미널 화면에 보낼 **ANSI 제어 바이트를 조립**하고, 화면이 얼마나 바뀌었는지와 글자가 몇 칸을 차지하는지 계산한다. TUI 에서 전체를 다시 그리지 않고 바뀐 곳만 고쳐 그릴 때 쓴다(37장).

> **이 모듈은 화면에 아무것도 쓰지 않는다**
>
> 바이트를 만들어 호출자 버퍼에 넣을 뿐이다. 모든 op 이 effects none 이다. 실제로 화면에 나가는 일은 cap io 를 가진 쪽(outbuf)이 한다 — 조립과 출력이 나뉘어 있다. 이것을 모르면 "코드는 도는데 화면에 아무것도 안 뜬다" 는 막다른 길에 빠진다. 그 대가로 이스케이프 조립과 diff 전부가 화면 없이 바이트 대조로 검증된다. 키 입력과 raw 모드는 tty 다.

```
use term as t .

let p option u64 . be t.goto buf 0 2 4 .
guard is_some p . else return 1 .
```

규약. 시퀀스 op 은 fmt 의 규약 그대로다 — (buf, pos, …) → option u64(새 pos). 앞 op 의 반환을 다음 op 의 pos 로 넘기면 시퀀스가 이어 붙고, 마지막 pos 가 곧 완성된 시퀀스의 길이다. **전량 아니면 무** — 자리가 모자라면 한 바이트도 쓰지 않는다. 반쯤 쓰인 이스케이프는 터미널이 뒷부분을 글자로 읽어 화면에 [3;5H 같은 찌꺼기를 띄우는, 조용히 틀린 출력이다. 좌표는 **0 기준**으로 받아 ANSI 의 1 기준으로 바꿔 낸다 — 1 기준은 선의 사정이다.

op	시퀀스	크기
clear buf pos	ESC[2J	4
goto buf pos row col	ESC[<row+1>;<col+1>H	4 + 두 수의 10 진 자릿수
sgr buf pos n	ESC[<n>m — 0 리셋 · 1 굵게 · 4 밑줄 · 7 반전	3 + n 의 자릿수
color256 buf pos n bg	ESC[38;5;<n>m(전경) · ESC[48;5;<n>m(배경)	8 + n 의 자릿수
cursor buf pos show	ESC[?25h · ESC[?25l	6

표 50.1 — 시퀀스 op 과 필요한 바이트

화면 diff. diff prev nxt w out pos 는 직전 화면 prev 와 새 화면 nxt(같은 길이, 폭 w 의 행들이 이어진 1 바이트 셀 화면)를 비교해, **같은 행 안에서 연속으로 바뀐 구간**마다 goto + 바뀐 바이트들 을 쓴다. 구간은 행 경계를 넘지 않는다. 바뀐 것이 없으면 0 바이트 — **0 바이트가 곧 "화면을 건드리지 마라"** 는 정상 답이다. 조건은 len prev == len nxt · w > 0 · len prev % w == 0 이다. UTF-8 화면은 행 단위로 diff_row_utf8 prev nxt row out pos 를 쓴다 — 바뀐 **코드포인트 구간**만 goto(row, 코드포인트 열) + 그 구간의 nxt 바이트로 낸다. 셀이 넓어지면 diff 의 뜻이 바뀌므로 diff 를 고치지 않고 별도 op 으로 옆에 두었다 — 옛 계약을 쓰던 코드가 조용히 달라지면 안 된다.

op	답	실패
diff prev nxt w out pos	some <새 pos>	조건 위반 · out 부족 = none, out **내용은 미정**
diff_row_utf8 prev nxt row out pos	some <새 pos>	유효하지 않은 UTF-8 · out 부족 = none, out **미정**

`row_cells row`	코드포인트 수	유효하지 않은 UTF-8 = `none`
`col_off row c`	코드포인트 c 번째의 바이트 오프셋	유효하지 않은 UTF-8 · 행이 c 셀에 못 미침 = `none`
`cp_width cp`	0 · 1 · 2 칸	실패 없음 — 모르면 1
`row_width row`	행의 칸 수 합	유효하지 않은 UTF-8 = `none`
`fit_width row cols`	`cols` 칸 안에 들어가는 **바이트 수** (글자 가운데를 자르지 않는다)	유효하지 않은 UTF-8 = `none` · 한 칸도 못 넣으면 `some 0`
`cluster_len s at`	`at` 에서 시작하는 글자 덩어리의 바이트 길이(1 이상)	범위 밖 · 경계가 아님 · 유효하지 않은 UTF-8 = `none`
`row_clusters row`	행의 글자 덩어리 수	유효하지 않은 UTF-8 = `none`

표 50.2 — diff 와 셈 op — 모두 `effects none`

넷을 헷갈리면 커서가 엉뚱한 데 선다. 바이트 수 ≠ 코드포인트 수(`row_cells`) ≠ 칸 수(`row_width`) ≠ 글자 수(`row_clusters`). `"한글"` 은 6 바이트 · 2 코드포인트 · **4 칸** · 2 글자이고, `"e"` + U+0301 은 3 바이트 · 2 코드포인트 · **1 칸** · **1 글자**다. `cp_width` 는 결합 표시 · 서식 문자(`unicode` 의 `is_zerowidth`)에 0, 한글 · CJK · 전각 기호에 2, 나머지에 1 이다. 폭은 못 맞혀도 1 이 쓸 만한 기본값이라 실패가 없다 — 속성 판정과 다른 점이다. 글자 덩어리는 선두 코드포인트 하나 + 뒤따르는 폭 0 코드포인트 전부다 — 백스페이스와 커서 이동을 이 단위로 해야 악센트만 지워지거나 커서가 글자 가운데 서지 않는다.

```
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
```

첫 diff 는 둘째 행 열 1 · 2 만 바뀌어 `ESC[2;2H`(6) + `xy`(2) = 8 바이트, UTF-8 행 diff 는 `ESC[1;2H`(6) + `라`(3) = 9 바이트다. 렌더 루프는 `nxt` 를 계산 → `diff` → `some p` 면 `subslice out 0 p` 를 `outbuf` 로 내보내고 `prev` 와 `nxt` 를 맞바꾼다. 맞바꾸기를 잊으면 같은 구간을 매번 다시 그린다.

> **반례. `diff` 가 `none` 인데 `out` 을 내보낸다**
>
> 두 diff 만 전량 아니면 무가 아니다 — 도중에 `out` 이 모자라면 쓴 바이트가 남는다. `none` 을 받고 그 버퍼를 흘려보내면 반쯤 쓰인 이스케이프가 나가 화면이 깨진다. 더 큰 버퍼로 다시 하거나 `clear` 뒤 전체를 다시 그린다.

> **반례. `prev` 와 `nxt` 를 뒤바꾼다**
>
> 같은 타입 · 같은 길이라 컴파일도 실행도 잡지 못하고, 바이트 수까지 같아 시험도 통과할 수 있다. 증상은 화면이 옛 내용으로 되돌아가는 것뿐이다. `diff <직전> <새것>` 순서를 외운다.

> **반례. 바이트 수로 자르거나 코드포인트 수로 칸을 센다**
>
> `subslice row 0 3` 은 3 바이트가 글자 경계라는 보장이 없어 반 글자가 나간다 — `fit_width` 가 경계를 지킨다. `row_cells "한글"` 은 2 인데 칸은 4 라, 코드포인트 수로 정렬하면 두 칸 어긋난다 — 칸은 `row_width` 로 묻는다.

주의. `none` 만 계속 나오면 버퍼 크기를 의심한다 — `pos` **뒤에 남은 자리**로 재야 한다. `out` 크기는 최악에 행 수 × goto 최대 폭 + 화면 크기로 어림한다. 무변경의 `some pos` 를 실패로 오해해 전체 다시 그리기로 물러나면 화면은 맞지만 매 프레임 깜빡인다. 좌표를 이미 1 기준으로 들고 있다면 그대로 넘기지 않는다 — 모든 것이 한 칸씩 밀린다. 1 바이트 셀 `diff` 에 UTF-8 화면을 넣으면 멀티바이트 문자 가운데를 자를 수 있다 — 두 diff 는 계약이 다른 별개 op 이다. **짓지 않은 것** — 국기(지역 표시 기호 쌍) · 이모지 ZWJ 시퀀스 · 한글 자모 조합 같은 나머지 글자 덩어리 규칙, 속성(색) 셀 diff. 부분 규칙을 완전한 척하기보다 무엇을 하지 않는지 적어 둔다.

sortlib — 제자리 quicksort

소스 lib/sort.low
층 L0 — 순수 계산
권한 없음

u64 슬라이스를 **제자리에서** 오름차순으로 정렬한다. 이진 탐색 전에(searchlib), 결과를 순서대로 보여 줄 때 쓴다. 공개 op 은 sort 하나다.

```
use sortlib .

sortlib.sort s .
```

> **성숙도 standard — 이 저장소의 첫 standard**
>
> 지금 있는 표면(sort 하나)을 **지킨다**. 이름 · 인자 · 의미가 바뀌려면 호환성 결정이 따로 든다. 더 자라지 않는다는 말은 아니다 — 비교자와 다른 원소 타입은 아직 없고, 생긴다면 **더하기**로 생긴다. 그리고 **성능은 약속하지 않는다** — 지금은 quicksort 다. 다른 원소 타입은 sortgen 이 맡는다.

정렬은 거의 모든 도구가 필요로 하는 **순수 계산**이라 세상에 닿지 않는다. 그러므로 빌트인이 아니라 라이브러리다 — Lowent 로 쓸 수 있으면 라이브러리다. 빌트인 증가 0 으로 지어졌다.

op	모양	실패
sort	proc (s mut slice u64) → void, effects none — 제자리 오름차순	없음(빈 슬라이스 · 한 원소도 무동작)
qsort(내부)	proc (s mut slice u64, lo0 u64, hi0 u64) → void	export 가 아니다 — 밖에서 부르면 E-VISIBILITY

표 50.1 — sortlib 의 op

깊이가 O(log n) 으로 묶인다 — 이것이 이 구현의 요점이다. 순진한 Lomuto 분할은 이미 정렬된 입력에서 피벗이 늘 끝이라 재귀가 O(n) 깊어지고, 네이티브 깊이 상한을 넘길 수 있다. 그래서 **작은 쪽을 재귀하고 큰 쪽을 루프한다**. 재귀에는 늘 절반 이하로 줄어드는 쪽만 쌓이므로 log n 을 넘지 못한다. 정확성이 아니라 자원 안전을 위한 것이다 — 느려질 수는 있어도 터지지는 않는다. 내부 qsort 는 반열린 구간 [lo0, hi0) 를 맡고, hi0 > len s 이거나 lo0 >= hi0 이면 아무것도 하지 않는다 — 뒤집힌 구간을 줘도 경계 밖을 만지지 않으므로 VM 과 네이티브가 같은 답(무동작)을 낸다.

새 배열을 돌려주지 않는 이유. 그러면 할당이 필요하고, 할당은 이 계층이 하지 않는 일이다. 비교자를 받지 않는 것도 같은 이유다 — 다른 순서가 필요하면 값을 미리 변환해 넣는다.

정렬 결과를 스스로 검증하는 모양 — 오름차순인가 **그리고** 합이 보존됐는가. 순서 결함과 원소 소실 · 복제 결함을 함께 잡는다.

```
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
```

> **반례. 부분 구간을 내부 op 으로 정렬한다**
>
> `sortlib.qsort s 0 (len s)` 는 E-VISIBILITY 다. 부분 구간은 `subslice` 로 잘라 `sort` 에 준다 — `sortlib.sort (subslice s 2 7) .`

> **반례. 불변 슬라이스를 넘기거나 반환값을 기대한다**
>
> `input s slice u64` 를 넘기면 타입 오류다 — 제자리 정렬이라 `mut slice u64` 여야 한다. 원본을 남겨야 하면 복사본을 정렬한다. `let r … be sortlib.sort s` 도 컴파일 에러다 — `output void` 이고 결과는 `s` 자신이다.

주의. 입력이 파괴된다. 안정 정렬이 아니다(같은 값끼리의 원래 순서를 보장하지 않는다 — `u64` 라 지금은 관측할 방법이 없지만 키 추출이 생기면 달라진다). 시간은 평균 $O(n \log n)$, 최악 $O(n^2)$이고, 추가 메모리는 $O(\log n)$ 재귀 프레임뿐이다. **확인하는 것** — 정렬됐는가와 순열인가(시험), 임의 입력에서 VM · 네이티브 일치(차등 시험).

sortgen — 제네릭 정렬(비교는 타입이 들고 온다)

소스 lib/sortgen.low
층 L0 — 순수 계산
권한 없음

원소가 u64 가 아니어도 정렬한다. 무엇이 더 작은지는 **타입이 스스로 말한다**. 레코드를 어떤 필드 기준으로 줄 세울 때 쓴다 — 원소가 그냥 u64 면 sortlib 이 더 간단하고 빠르다. 세 가지가 한 벌이다 — ① 타입이 ordered 를 충족한다고 선언하고 ② 그 타입의 less 를 쓰고 ③ 타입을 인자로 주며 부른다.

```
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
```

왜 비교 함수를 값으로 넘기지 않나. 흔한 답(C 의 qsort, C++ 의 comparator)은 비교 함수를 값으로 넘기는 것이다. 이 언어는 그 길을 쓰지 않는다. 일급 함수가 없고, 있어도 **간접 호출**이 생긴다 — 호출 자리를 보면 공짜처럼 보이는데 함수 포인터를 거치는 비용이 숨는다. 이 언어의 답은 **comptime 타입 파라미터 + trait 경계**다 (22장, 23장). 호출마다 그 타입 전용 op 이 만들어지고 비교는 직접 호출로 박힌다. 간접 호출 0, 비교자를 담을 공간 0.

op	모양	실패
ordered	trait — less (self, self) → bool	미충족 = E-BOUND-UNSAT(컴파일)
sort_by	(comptime t, mut slice t) → void, requires ordered t — 삽입정렬(안정)	없음
sort_fast	같은 모양 — quicksort, 큰 배열용	없음
lower_by	(comptime t, slice t, key t) → u64	없음 — 없으면 삽입점(len s 가능)
find_by	(comptime t, slice t, key t) → option u64	없으면 none

표 50.1 — sortgen 의 op

less a b 가 참이면 a 가 b 보다 앞에 온다. 이 관계는 **엄격한 약한 순서**여야 한다 — 특히 less a a 는 거짓이어야 한다. requires ordered t 가 없으면 less 가 있는지 아무도 묻지 않은 채 본문이 그것을 부른다. 경계가 있으면 없는 타입으로 부르는 순간 컴파일이 거절한다.

sort_fast 이야기. 처음엔 분할이 피벗의 **복사**를 요구한다고 보고 제네릭 quicksort 를 짓지 않았다. 틀렸다 — 피벗을 **자리**(인덱스)로 들고 비교할 때마다 다시 읽으면 된다. 분할 루프 동안 피벗은 hi - 1 에 가만히 있고, 옮기는 일은 전부 swap 이 한다.

반례. `satisfies` 를 빠뜨리거나 `u64` 를 그대로 넘긴다

`satisfies sortgen.ordered .` 가 없는 구조체나 `u64` 로 `sort_by` 를 부르면 `E-BOUND-UNSAT` 이다. `u64` 는 메서드를 붙일 자리가 없어 trait 을 충족할 수 없다 — 필드 하나짜리 구조체로 감싸면 레이아웃은 그대로 8바이트이고, 아니면 `sortlib.sort` 를 쓴다.

반례. `less` 가 같은 값에 참을 낸다

`return le (field a key) (field b key) .` 는 컴파일은 통과하지만 같은 값이 둘 이상이면 루프가 끝나지 않을 수 있다. 비교는 `lt` 여야 한다.

주의. 내림차순 · 다중 키는 `less` 를 그렇게 쓰면 된다 — 모드 인자를 다는 것이 곧 엔트로피다. 타입마다 코드가 하나씩 생긴다(단형화의 대가, 대신 간접 호출이 없다). **읽으면 뷰다** — `let a t be index s j` 는 복사가 아니라 그 자리를 가리키는 창이라, 원소를 맞바꿀 때 `swap` 을 쓴다(`set` 두 번으로는 자기 자신을 덮어쓴다). 직접 정렬을 짠다면 조심한다 — 한 번 맞바꾸고 나면 같은 이름 `a · b` 가 다른 값을 본다. 한 걸음에 판정은 한 번만 한다. 두 번 비교하면 두 번째가 이미 바뀐 것을 보고 원소가 한 칸만 내려가 멈추는 부분 정렬이 되고, 원소 셋에 단일 키면 답이 우연히 맞아 결함이 숨는다(실제로 그렇게 숨었다가 다중 키 예제에서 드러났다). `sortlib` 과 갈라져 있는 것은, 제네릭 틀이 들어오면 `sortlib` 을 쓰는 모든 파일이 일부 대조 검사에서 빠져야 했기 때문이다.

searchlib — 정렬된 슬라이스의 이진 탐색

소스 lib/search.low
층 L0 — 순수 계산
권한 없음

정렬된 슬라이스에서 값을 이진 탐색으로 찾는다. 가운데를 찍어 보고 찾는 값이 왼쪽인지 오른쪽인지 정한 뒤 남은 절반만 다시 뒤지므로, 원소가 백만 개여도 스무 번쯤이면 끝난다. 대신 미리 줄이 서 있어야 방향을 정할 수 있다. sortlib 의 짝이고, 둘이 함께 **정렬된 집합 · 맵**을 만든다 — 해시맵과 달리 순서를 지키므로 범위 질의와 "다음 큰 값" 이 된다. 파일 이름은 search.low 이지만 모듈 이름은 searchlib 이다.

```
use searchlib .
use sortlib .

sortlib.sort s .
let i option u64 . be searchlib.bsearch s 42 .
```

정렬을 검사하지 않는다

s 가 오름차순이라는 전제는 **호출자의 몫**이다. 전제가 깨지면 답이 틀리지만 멈추지는 않는다 — 범위 밖을 만지지 않으므로 멈출 일이 없고, 대신 **조용히 틀린 답**이 나온다. "분명히 있는 값인데 none" 이면 십중팔구 정렬을 하지 않은 것이다.

op	모양	답
bsearch	(s slice u64, target u64) → option u64	target 의 인덱스, 없으면 none. 중복이면 그중 **하나** (어느 것인지 규정하지 않는다)
lower_bound	(s slice u64, target u64) → u64	target 이상인 첫 인덱스. 모두 작으면 len s

표 50.1 — searchlib 의 op — 모두 fn · effects none

고르는 법 — "있나 없나, 있으면 어디" 는 bsearch, "여기 끼우면 어디" 또는 "이 값 이상이 어디부터" 는 lower_bound 다. lower_bound 는 **삽입점**이자 **범위의 시작**이다 — [lower_bound s a, lower_bound s b) 가 a ≤ x < b 인 원소들의 인덱스 구간이다. option 이 아닌 까닭은 "못 찾음" 이 없는 물음이기 때문이다(끝 인덱스도 답이다). upper_bound · 삽입 · 삭제는 없다 — 하한 하나로 포함 여부 · 삽입점 · 범위 질의가 다 선다.

구현은 반열린 구간 [lo, hi) 로 돌고 중점은 lo + (hi - lo)/2 로 잡는다 — (lo + hi)/2 는 큰 인덱스에서 넘칠 수 있다(고전적 결함).

```
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
```

반례. bsearch 의 none 을 가르지 않는다

return some_value r . 는 컴파일은 통과하고, 못 찾은 순간 E-VM-NONE 으로 멈춘다. 있는 값만 넣어 시험하면 걸리지 않는다 — option 을 보면 guard is_some 을 붙인다(11장).

> **반례. 중복 원소에서 `bsearch` 인덱스에 뜻을 싣는다**
>
> `[5,5,5]` 에서 `bsearch s 5` 는 $0 \cdot 1 \cdot 2$ 중 어느 것이든 낼 수 있다. "첫 5 의 자리" 가 필요하면 `lower_bound s 5` 다.

> **반례. `lower_bound` 결과로 곧장 `index` 한다**
>
> 못 찾으면 `i` 가 `len s` 이고 그 자리는 없는 자리다 — `index s i` 가 `E-VM-BOUNDS` 로 멈춘다. 값이 표의 최댓값보다 클 때만 터지므로 실제 데이터에서 걸리기 쉽다. `guard lt i (len s) .` 를 먼저 둔다.

주의. 내림차순 자료에는 그대로 쓸 수 없다(비교자를 바꿔 끼울 자리가 없다). **정렬은 한 번, 탐색은 여러 번** — 찾을 때마다 `sort` 를 부르면 이득이 통째로 사라진다. 정렬 뒤에 원소 하나를 `set` 으로 바꾸면 다시 정렬해야 한다. 하한 둘의 차를 뺄 때는 순서를 지킨다 — `sub` 이 아래로 넘친다. 순수 op 이라 같은 슬라이스를 여러 곳에서 동시에 탐색해도 된다(쓰는 곳이 없을 때). 다른 원소 타입은 `sortgen` 의 `lower_by` · `find_by` 다.

hashmap — u64 → u64 해시 맵

소스 lib/hashmap.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

u64 열쇠로 u64 값을 넣고 찾는 해시 맵이다. 번호로 무언가를 빨리 찾아야 할 때(아이디 → 개수, 핸들 → 상태) 쓴다. 문자열 열쇠는 strmap, 원소 타입과 얼로케이터를 고르는 컨테이너는 mapgen 이다(34장).

```
let stored bool be hashmap.put b 7 42 .
let v option u64 . be hashmap.lookup b 7 .
```

꼭 알아 둘 것 둘

> ① 맵은 스스로 메모리를 만들지 않는다 — 저장 공간 b 를 호출자가 마련해 매번 넘긴다. ② 새 맵은 **반드시 전부 0 으로** 시작한다. 이 둘을 어기면 에러 없이 답이 틀린다 — 쓰레기 값이 든 슬라이스는 "이미 뭔가 들어 있는 맵" 으로 읽혀, 넣은 적 없는 키가 찾히거나 방금 넣은 키가 none 으로 나온다.

오픈 어드레싱. 키마다 연결 리스트를 달지 않고, 자리가 차 있으면 같은 표의 다음 슬롯으로 옮겨 가며 넣는다(선형 프로빙) — 버퍼 하나로 끝나 호출자 소유 규율과 맞는다. 백킹은 mut slice u64 하나이고 [key0+1, val0, key1+1, val1, …] 로 눕는다(슬롯 수 N = len / 2). key 슬롯에 k+1 을 저장하므로 0 이 빈칸 표식이 되고 **키 0 도 저장할 수 있다**. 삭제는 key 슬롯을 MAX_U64 인 **묘비**로 덮는다 — 빈칸으로 되돌리면 충돌로 뒤에 밀려 들어간 키들의 자취가 끊겨 그 뒤 키를 못 찾게 된다. lookup 은 묘비를 건너뛰고, put 은 재사용하고, 순회는 세지 않는다. 해시는 곱셈 해시(황금비 상수)다.

op	모양	실패
put	proc (b mut slice u64, k u64, v u64) → bool	꽉 참 · 예약 키 → false
lookup	proc (b slice u64, k u64) → option u64	없음 · 예약 키 → none
del	proc (b mut slice u64, k u64) → bool	없던 키 · 예약 키 → false
size	fn (b slice u64) → u64	슬롯 수(항목 수가 아니다)
occupied_at	fn (b slice u64, slot u64) → bool	범위 밖 → false
key_at · val_at	fn (b slice u64, slot u64) → u64	occupied 일 때만 뜻이 있다
rehash	proc (nb mut slice u64, ob slice u64) → bool	nb 가 작으면 도중 false

표 50.1 — hashmap 의 op — 모두 effects none

- put 은 같은 키면 제자리 갱신이다. 프로브 중 처음 만난 묘비를 기억해 뒀다가 키가 뒤에 없음이 확인되면 거기 넣는다 — 묘비 재사용으로 클러스터가 자라지 않는다.
- **자동으로 자라지 않는다.** 꽉 차면 put 이 false 를 낸다(멈추지 않는다). 자라려면 호출자가 큰 백킹을 0 으로 마련해 rehash nb ob 를 부른다. 살아 있는 항목만 옮기므로 묘비도 사라진다(압축). 옛 백킹은 읽기만 하므로 실패해도 온전하다.
- **키 둘을 담지 못한다.** MAX_U64 와 MAX_U64 - 1 은 인코딩에 예약됐다 — 별도 상태 바이트 없이 삭제를 얻는 대가다.
- 순회는 0 .. size 를 훑으며 occupied_at 인 슬롯만 처리한다. 순서는 해시 순서다 — 정렬된 열거가 필요하면 키를 뽑아 sortlib 로 정렬한다.

```
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
```

> **반례. `occupied_at` 없이 슬롯을 읽는다**
>
> 빈칸 · 묘비의 `key_at` · `val_at` 은 뜻 없는 값이라(빈칸의 `key_at` 은 `0 - 1` 까지 간다) 에러 없이 합계가 틀린다. 순회는 언제나 `occupied_at` 으로 거른다.

> **반례. `put` 의 `false` 를 보지 않는다**
>
> 꽉 찬 맵은 멈추는 대신 `false` 를 내므로 그 줄은 조용히 지나가고, 나중에 `lookup` 이 `none` 을 내어 검사 없는 `some_value` 에서 `E-VM-NONE` 으로 멈춘다. 멈추는 자리가 원인에서 멀다 — `guard eq (hashmap.put …) true .` 가 습관이어야 한다.

주의. op 이름이 `get` 이 아니라 `lookup`, `capacity` 가 아니라 `size` 인 이유 — `get` 은 페이로드 enum 해체 낱말이고 `capacity` 는 `alloc_bytes … capacity n` 의 문법 낱말이라 op 이 가질 수 없다. 넣고 지우기를 되풀이하면 묘비가 쌓여 프로브가 길어진다 — 같은 크기라도 `rehash` 로 옮겨 압축한다. 백킹 길이가 홀수면 마지막 u64 하나는 쓰이지 않는다.

strmap — 문자열 키 해시 맵

소스 lib/strmap.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

바이트열(문자열) 열쇠로 u64 값을 넣고 찾는 해시 맵이다 — 단어 세기 · 심볼 테이블 · 설정 맵. hashmap 을 가변 길이 바이트열 키로 넓힌 것이다. 키 바이트는 넣을 때 아레나로 **복사**되므로 원본 문자열의 수명에 매이지 않는다.

```
let stored bool be strmap.put slots keys "apple" 1 .
let v option u64 . be strmap.lookup slots keys "apple" .
```

버퍼 둘을 호출자가 마련한다. 처음에 slots 는 전부 0 이어야 한다.

- slots(mut slice u64) — 슬롯 표. slots[0] 은 **아레나 커서**(키 바이트가 얼마나 쓰였나)이고, 진짜 슬롯은 인덱스 1 부터 슬롯 하나 = u64 셋 [keyoff, keylen, value] 다. 슬롯 수 N = (len slots - 1) / 3.
- keys(mut slice u8) — 키 바이트 아레나. 여러 키의 바이트를 한 버퍼에 이어 담는다(커서를 앞으로만 미는 bump).

keylen 은 세 뜻이다 — 0 = 빈칸, MAX_U64 = 묘비, 그 밖 = 살아 있는 키의 길이. 실제 키 길이는 MAX 일 수 없으므로 예약 없이 묘비 값이 된다. 해시는 FNV-1a 64 + 선형 프로빙이다.

op	모양	실패
put	proc (slots mut slice u64, keys mut slice u8, k slice u8, v u64) → bool	꽉 참 · 아레나 부족 · 빈 키 → false
lookup	proc (slots slice u64, keys slice u8, k slice u8) → option u64	없음 · 빈 키 → none
del	proc (slots mut slice u64, keys slice u8, k slice u8) → bool	없던 키 · 빈 키 → false
size	fn (slots slice u64) → u64	len slots < 4 → 0
occupied_at	fn (slots, slot u64) → bool	범위 밖 → false
keylen_at · keyoff_at · val_at	fn (slots, slot u64) → u64	범위 밖 → 0
rehash	proc (ns mut slice u64, na mut slice u8, os slice u64, oa slice u8) → bool	ns · na 가 작으면 도중 false

표 50.1 — strmap 의 op — 모두 effects none

- put 은 같은 키(길이 + 바이트 일치)면 값만 갱신한다 — 새 슬롯도 아레나도 쓰지 않는다. 새 키는 아레나에 복사된다. 프로브 중 처음 만난 묘비를 재사용한다.
- del 은 keylen 을 묘비로 덮고 value 를 0 으로 지운다. **아레나의 옛 키 바이트는 남는다** — 회수는 rehash 가 새 아레나로 옮기며 한다(압축).
- 키 바이트는 subslice keys off (add off klen) 으로 읽는다. 그 뷰는 아레나의 뷰라 아레나가 살아 있는 동안만 유효하다.

> **반례. 빈 문자열 키**
>
> strmap.put slots keys "" 7 은 언제나 false 다. keylen 0 이 빈칸 표식이라 빈 키는 인코딩상 존재할 수 없다. lookup "" 도 언제나 none 이다.

> **반례. 삭제 뒤 `keylen > 0` 으로 순회한다**
>
> 묘비의 keylen 은 MAX 라 `gt keylen 0` 이 참이다 — 지운 항목이 집계에 섞여 든다. `del` 을 한 번이라도 쓴 맵은 `occupied_at` 으로 거른다.

> **반례. `slots[0]` 을 직접 만진다**
>
> 아레나 커서를 손대면 다음 `put` 이 키 바이트를 엉뚱한 곳에 복사한다. 멈추지 않고 조용히 틀린다. 초기화는 “전부 0”, 이후 수정은 op 으로만 한다.

주의. 아레나는 bump 다 — 서로 다른 키를 넣었다 지웠다 되풀이하면 슬롯이 남아도 아레나 부족으로 `put` 이 `false` 가 될 수 있다. `put` 의 `false` 하나로 슬롯 부족과 아레나 부족이 구별되지는 않는다. 버퍼 크기는 슬롯 N 개 = `3N + 1` u64, 아레나는 살아 있는 키 길이 합 + 죽은 바이트로 짐작한다. 실제 쓰임은 파일을 읽어 `strings` 로 단어를 쪼개고 빈도를 세는 프로그램이다(34장).

vecs — 호출자가 버퍼를 드는 성장 바이트 벡터

소스 lib/vec.low
층 L1 — 얼로케이터를 using 으로
권한 없음 — 얼로케이터를 건네받는다

자리가 모자라면 **더 큰 자리를 받아 옮기는** 바이트 벡터다. 얼마나 커질지 미리 모르는 바이트열을 모을 때 쓴다. 없는 것은 재할당이지 성장이 아니다 — 더 큰 자리를 받아 옛 내용을 옮기면 되고, 그것은 Lowent 로 쓸 수 있다. 그래서 빌트인이 아니라 라이브러리이고, 성장 정책(두 배)도 라이브러리의 몫이다.

	vecs	vecgen
저장소	**호출자가 든다** — op 에 버퍼를 넘긴다	컨테이너가 소유한다
성장	새 자리를 **값으로 돌려준다**(호출자가 다시 묶는다)	안에서 갈아 끼운다
얼로케이터	부를 때마다 using 으로 건넨다	필드로 들고 있다

표 50.1 — vecs 와 vecgen — 다른 것은 소유다

region 안에서 버퍼 수명을 **직접** 쥐고 싶을 때가 vecs 다. 벡터는 vec_u8(길이 len 하나)과 mut slice u8(저장소) **두 값의 쌍**으로 산다. 용량은 따로 들지 않는다 — len buf 가 곧 용량이다. 불변식 v.len ≤ len buf 를 push 가 유지한다.

op	모양	실패
vec_u8	구조체 — len u64	—
next_cap	fn (cur u64) → u64 — 0 이면 8, 아니면 두 배	없음
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	OOM 이면 none — 이때 v · buf 는 그대로

표 50.2 — vecs 의 op

push_byte 는 자리가 있으면 그대로 쓰고 some buf 를, 꽉 찼으면 send al reserve (next_cap (len buf)) 로 두 배를 받아 옛 내용을 복사한 뒤 some <새 버퍼> 를 돌려준다. 얼로케이터는 위치 인자가 아니라 using 절로 건넨다 — 타입 a 는 그 출처에서 채워지고 단형화되므로 전달 비용은 0 이다(35장). 몰래 할당하지 않으므로 얼로케이터를 가지지 않은 코드는 이 op 을 부를 수 없다.

```
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
```

2 바이트로 시작해 10 개를 밀며 2 → 4 → 8 → 16 으로 세 번 자라고, `ABCDEFGHIJ` 를 출력한다.

> **반례. 돌려받은 버퍼로 다시 묶지 않는다**
>
> `set buf (some_value r) .` 를 빼먹으면 컴파일도 실행도 통과하는데 결과가 소리 없이 틀린다. 성장이 일어난 뒤 옛 `buf` 를 계속 주면 `v.len` 은 새 버퍼 기준으로 올라가 있는데 쓰기는 옛(작은) 버퍼에 간다. **반환값이 곧 다음 버퍼다** — 이 모듈에서 가장 위험한 실수다.

> **반례. OOM 을 보지 않는다 · 순수 fn 에서 부른다**
>
> `some_value r` 를 검사 없이 꺼내면 얼로케이터가 말랐을 때 `E-VM-NONE` 으로 멈춘다. `effects none` 인 `fn` 에서 부르면 `E-EFFECT-CALC` 다 — `push_byte` 는 `effects state` 인 proc 이다. trait 을 충족하지 않는 값(예: `u64`)을 출처로 건네면 `E-BOUND-UNSAT` 이다.

주의. 성장은 복사다 — push 한 번이 최악 O(len)이지만 두 배 정책이라 상환 O(1)이다. 크기를 미리 알면 처음부터 그만큼 받는 쪽이 싸다. **옛 버퍼는 회수되지 않는다** — 범프 위에서 n 까지 키우면 뒷받침 소비는 대략 2n 이고, 옛 버퍼는 `region` 이 끝날 때 통째로 걷힌다(18장). 머리와 버퍼는 쌍이다 — 서로 다른 벡터의 것을 섞어 넘기는 것을 막을 장치는 없다. 원소 타입 일반화 · pop · insert · remove · 줄어들기는 없다.

spsc — 락 없는 SPSC 링 버퍼

소스 `lib/spsc.low`
층 L0 — 순수 계산(호출자의 저장)
권한 `cap atomic`

스레드 사이로 값을 넘기는 가장 단순한 방법은 락이다. 그런데 생산자가 하나, 소비자가 하나뿐이면 락이 필요 없다 — 생산자는 `tail` 만 쓰고 소비자는 `head` 만 쓰며, 서로의 것은 원자적으로 **읽기만** 한다. 이것이 SPSC(single-producer single-consumer)이고, 락 없는 구조 중 정직하게 짧은 거의 유일한 것이다.

> **생산자 하나 · 소비자 하나가 계약이다**
>
> 두 스레드가 `push` 하거나 두 스레드가 `pop` 하면 이미 SPSC 가 아니고, 이 모듈은 그것을 막지 못한다. 그래서 `peek` · `drain` 같은 편의 op 을 넣지 않았다 — 편의 op 이 늘수록 그 계약을 어기는 방법이 늘어난다. 표준 라이브러리에서 작은 표면은 게으름이 아니라 계약을 지키는 수단이다. MPSC · MPMC · seqlock 은 없다 — CAS 재시도와 ABA 가 붙으면 검증되지 않은 코드를 표준으로 만드는 일이 된다.

한 칸을 늘 비워 둔다. 그래서 실제 용량은 `len buf - 1` 이다. 그 대가로 `head == tail` 이 **언제나** "비었다"를 뜻하고 가득 참과 헷갈리지 않는다.

op	하는 일	돌려주는 것
`spsc_capacity buf`	담을 수 있는 최대 개수	`len buf - 1`
`spsc_push k ctl buf v`	밀어 넣기(**생산자 전용**)	1 성공 · 0 가득 참
`spsc_pop k ctl buf out`	꺼내기(**소비자 전용**), 값은 `out[0]` 에	1 성공 · 0 비었음
`spsc_count_about k ctl buf`	지금 개수(관찰값)	개수

표 50.1 — `spsc` 의 op

`ctl` 은 길이 2 이상의 `mut slice u64` 로 `[0]` = head(소비자), `[1]` = tail(생산자)이다. `k` 는 `cap atomic` — 공유 자리를 원자적으로 만질 권한이다. `count_about` 에 `about` 이 붙은 이유 — 읽는 순간 이미 달라졌을 수 있다. 이름이 그것을 말하지 않으면 쓰는 사람이 그 값을 믿는다.

무엇이 이 코드를 받치나. 두 가지 증명이다. 첫째, 색인을 `mod … (len buf)` 로 감싸면 언제나 범위 안이다 — `lowentc --emit-proof lib/spsc.low` 가 링 버퍼의 두 접근(`index`, `index.store`)과 색인을 올리는 두 덧셈의 검사가 증명으로 사라졌다고 적는다(41장). 둘째, 데이터 쓰기 → 색인 공개 → 색인 관찰 → 데이터 읽기가 약한 메모리 모델(RC11)에서 데이터를 나른다 — 그리고 알고리즘 자체의 정확성(성공하면 자원이 버퍼로 넘어가고 실패하면 되돌아온다)은 기존 증명(gpfsl 의 `circ_buff`)을 빌렸다. 우리 `spsc_push` 가 그 증명의 코드와 한 줄씩 같다는 대응은 사람이 읽어 확인한 것이고 기계 검증되지 않았다(47장).

순서(ordering)는 필요한 만큼만 세다. 생산자가 head 를 읽을 때는 `acquire`(소비자가 공개한 것), 자기 tail 을 읽을 때는 `relaxed`(내가 쓴 값), tail 을 공개할 때는 `release`(앞의 데이터 쓰기가 먼저 보여야 한다)다. 소비자 쪽은 대칭이다. 넷 중 동기화가 필요한 자리는 둘뿐이다. 이 표기(`order <이름>`)는 한때 파서 결함으로 한 번도 닿지 못했고, 이 라이브러리를 쓰려다 드러나 고쳤다 — 쓰지 않는 기능은 있어도 없는 것이다.

확인하는 것 — 빈 것에서 꺼내기 실패, 가득 참에서 넣기 실패, FIFO, 링을 한 바퀴 넘겨도 그대로. 그 시험은 단일 스레드다(VM 이 단일 스레드다). 진짜 두 스레드에서의 메모리 순서의 정당성은 실행이 아니라 증명이 준다 — 둘을 섞지 않는다.

hash — 해시(해시맵 자리 · 손상 검출 · SHA-256)

소스 lib/hash.low
층 L0 — 순수 계산
권한 없음

바이트열에서 수를 낸다. 성격이 다른 셋이 한 모듈에 있고, 섞으면 위험한 쪽으로 틀린다.

> **of 와 crc 는 암호용이 아니다**
>
> of(FNV-1a 64)는 **해시맵 자리** — 빠르고 잘 흩어지지만 암호용이 아니다. crc(반사 CRC-32)는 **우연한 손상** 검출 — 원하는 CRC 를 맞추는 것은 쉬우므로 위조를 막지 못한다. digest · digest_ok(SHA-256)는 암호 해시로, 무결성 증명은 이쪽이다. 키가 있는 인증은 digest 도 아니고 hmac 이다 — 해시 하나로는 **누가 보냈는가** 에 답할 수 없다.

```
use hash .

fn slot input key slice u8 . input nslots u64 . output u64 . do
  return hash.bucket_of key nslots .
end
```

왜 알고리즘을 고정했나. 해시는 그럴듯한 수가 나오면 맞아 보인다. 고정하지 않으면 검사할 것이 남지 않는다. 고정했기에 표준이 정한 검사값이 시험의 기준이 되고 (CRC-32 의 "123456789" → 0xCBF43926), VM 과 네이티브가 갈리면 그 자리에서 드러난다.

op	모양	쓸모
of	(data slice u8) → u64	FNV-1a 64 — 해시맵 자리
bucket_of	(data, nslots u64) → u64	칸 고르기(나머지 연산)
bucket_mask	(data, mask u64) → u64	칸 수가 2 의 거듭제곱일 때
crc	(data slice u8) → u64	반사 CRC-32 — 손상 검출
crc_ok	(data, want u64) → bool	검사값 대조
maybe_same	(a, b slice u8) → bool	해시가 같은가 — 같다는 증명이 아니다
digest	(data, out mut slice u8) → u64	SHA-256 — 쓴 바이트 수(32)
digest_ok	(data, out mut slice u8, want slice u8) → bool	받은 다이제스트와 대조

표 50.1 — hash 의 op

digest_ok 가 있는 이유 — 해시의 올바른 쓰임은 왕복이 아니라 **대조**다. 다시 계산해 받은 것과 맞대는 것이 유일하게 옳은 쓰임이라 그 모양이 op 으로 있다(want 가 32 바이트가 아니면 그대로 false — 길이부터 계약(op 이 스스로 적는 약속)이다). maybe_same 에 maybe 가 붙은 이유 — 해시가 같아도 값이 같다는 뜻은 아니다. same 이라 불렀으면 이름이 거짓말이 됐을 것이다.

짓지 않은 것 — 키드 해시(그 자리는 hmac), 시드 가능한 해시(HashDoS 방어), 스트리밍(조각으로 이어 먹이기), SHA-256 말고 다른 암호 해시.

`math` — 부동소수 수학

소스 `lib/math.low`
층 L0 — 순수 계산(호스트 전용)
권한 없음

`sqrt·sin·exp` 같은 내장 연산 위에 자주 쓰는 것들을 얹는다 — 비교(`close`), 상수(`pi·e`), 각도 변환, 빗변, 밑이 있는 로그, 선형 보간. 권한은 없지만 C 수학 라이브러리에 붙으므로 **호스트 전용**이다 — 운영체제 없는 대상에는 부동소수 자체가 없을 수 있다(`cortex_m` 은 `no_float`).

> **`==` 로 비교하지 않는다 — `close` 를 쓴다**
>
> ```
> guard math.close (math.hyp 3.0 4.0) 5.0 0.000001 . else return 1 .
> ```
>
> 부동소수에서는 "같다" 가 잘 정의되지 않는다(4장). 그래서 이 모듈은 `close(a, b, tol)` 을 주고, 라이브러리 자신의 시험도 그것으로 판정한다.

확인이 비트가 아닌 까닭. C 수학 라이브러리는 **구현마다 마지막 자리가 다를 수 있다**. 정확한 비트를 주장하면 다른 기계와 다른 libc 에서 도구가 멀쩡한데 시험이 실패하는 거짓 실패가 난다. 그래서 이 모듈의 확인은 **알려진 답**(`sin(0) = 0`)과 **항등식**(`sin²+cos² = 1`, `exp(log x) = x`)이고 판정은 `close` 로 한다. 얼마나 가까우면 같다고 볼지(`tol`)는 부르는 쪽의 문제이지 라이브러리가 정할 것이 아니다.

op	하는 일
`close`	`¦a − b¦ ≤ tol` — 이 모듈의 비교 방식
`pi · e`	상수
`deg_to_rad · rad_to_deg`	각도 변환
`hyp`	`sqrt(x² + y²)`
`log_base`	밑이 있는 로그(`log x / log b`)
`lerp`	선형 보간 `a + (b − a)·t`

표 50.1 — `math` 의 op

내장 연산으로 바로 쓰는 것 — `sqrt·abs·floor·ceil·round·sin·cos·exp·log·pow`.

짓지 않은 것 — `atan2·asin·acos·tan·log2·log10·cbrt`, 복소수, 고정소수, `f32` 전용 얼굴. 필요해지면 내장 연산 하나와 이 파일 한 줄로 붙는다. `hyp` 는 `sqrt(x²+y²)` 를 그대로 셈한다 — libm 의 `hypot` 처럼 넘침을 피하는 재조정을 하지 않으므로 아주 큰 값에서는 다르게 답할 수 있다.

random — 난수(재현되는 열 · OS 엔트로피)

소스 lib/random.low
층 L0 — 순수 계산 · OS 엔트로피는 권한으로
권한 bytes · seed_from_os 에 cap random

> **둘을 섞지 않는다 — 이 모듈의 전부가 이 경고다**
>
> step · below_biased · coin 은 **splitmix64** 로 만든 재현 가능한 열이다. 권한이 없고 시험 · 시뮬레이션 · 셔플에 쓴다. bytes · seed_from_os 는 **OS 엔트로피**다. cap random 이 필요하고 열쇠 · 논스 · 토큰에 쓴다. 시험은 재현되어야 하고 열쇠는 예측되면 안 된다 — 한 낱말로는 둘 다 못 하므로 이름을 갈라 두었다. step 으로 만든 값을 열쇠로 쓰면 시드를 아는 사람이 그 열쇠를 알고, 시드는 보통 코드나 로그에 남는다.

재현되는 쪽. 상태(= 시드)를 받아 다음 상태를 돌려준다. 호출자가 상태를 들고 다니므로 같은 시드는 언제나 같은 열을 낸다.

```
var s u64 be 12345 .
set s (random.step s) .
let c bool be random.coin s .
```

예측할 수 없는 쪽. cap random 을 받아야 부를 수 있다. 채운 바이트 수를 답하고 **못 채우면 0** 이다 — 0 을 받고 그대로 쓰면 초기화되지 않은 버퍼를 열쇠로 쓰는 것이다.

```
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	모양	비고
step	(seed u64) → u64	splitmix64 — 알고리즘 고정(고정해야 검사값이 기준이 된다)
below_biased	(seed u64, bound u64) → u64	이름이 **편향을 자백한다** — 균등이 필요하면 이것이 아니다
coin	(seed u64) → bool	앞 · 뒤
bytes	(cap random, dst mut slice u8) → u64	채운 바이트 수. **0 = 실패**
seed_from_os	(cap random, scratch mut slice u8) → u64	OS 에서 받은 시드 하나 — 재현되는 열의 출발점을 예측할 수 없게. scratch ≥ 8

표 50.1 — random 의 op

below_biased 의 이름에 biased 가 있는 이유 — 나머지 연산으로 범위를 줄이면 작은 값이 조금 더 자주 나온다. 그것을 감추지 않고 이름에 적었다.

실패를 불러와 시험한다. 환경 변수 LOW_HOST_FAULT="random:err" 는 엔트로피를 받지 못하게(→ 0), LOW_HOST_FAULT="random:short=3" 은 3 바이트만 채우게 한다. **부분 채움이 조용히 무서운 쪽이다** — 버퍼 절반이 예전 값인 채로 열쇠가 된다. 그래서 채운 수를 확인하는 것이 계약이다(28장).

짓지 않은 것 — 균등 범위 난수(거절 표본추출), 셔플, 분포(정규 등), 재현되는 열의 직렬화, 암호용 CSPRNG(bytes 는 OS 에 묻는 것이지 이 모듈이 만드는 것이 아니다).

hmac — HMAC-SHA256 과 HKDF

소스 lib/hmac.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

두 가지를 한다. **HMAC** 은 키를 아는 사람만 만들 수 있는 증표(mac)로 "누가 보냈는가" 를 답한다(RFC 4231). **HKDF** 는 하나의 비밀에서 여러 키를 갈라내는 표준 방법(extract + expand1)이다(RFC 5869).

> **무엇을 약속하고 무엇을 하지 않나**
>
> 상수 시간을 약속하지 않고 감사받지 않았다. **MAC 비교는 부르는 쪽의 몫이다** — 이른 반환 없이(바이트를 XOR 로 누적해) 비교한다. 먼저 다른 바이트에서 돌아오는 비교는 태그를 한 바이트씩 알아내게 해 준다. expand1 은 **한 블록(32 바이트 이하)만** 낸다.

왜 해시로는 안 되나. hash 의 digest(SHA-256)는 "내용이 그대로인가" 만 답한다. 누구나 다시 계산할 수 있으므로 누가 보냈는지는 답하지 못한다. 키가 들어가야 그 물음이 열린다. 뒷받침(scratch)은 호출자가 든다. 크기 요구가 문서가 아니라 **가드**로 박혀 있어서, 모자라면 0 을 답하고 아무것도 쓰지 않는다.

op	하는 일	요구
mac	HMAC-SHA256(key, msg) → out 32 바이트	out ≥ 32 · scratch ≥ 64 + len(msg)
key_block	K′(64 바이트로 맞춘 키)를 만든다	pad ≥ 64. 64 보다 긴 키는 해시로 줄인다
extract	HKDF-Extract: PRK = HMAC(salt, ikm)	mac 과 같다
expand1	HKDF-Expand 의 첫 블록: T(1) = HMAC(prk, info ‖ 0x01)	scratch ≥ 130 + len(info) · len(info) ≤ 60

표 50.1 — hmac 의 op

왜 expand1 은 한 블록뿐인가. TLS 1.3 의 HKDF-Expand-Label 은 32 바이트 이하만 쓴다(키·IV·finished 모두). 그래서 T(2) 를 잇는 반복을 짓지 않았다. **왜 60 에서 자르나.** msg 가 scratch 안(128 부터)에 살고 mac 이 scratch[0 .. 64+len(msg)] 를 덮어쓰므로, info 가 길면 둘이 겹쳐 해시가 자기 입력을 밟는다. 겹치는 순간을 가드가 거절한다 — 조용히 틀리느니 거절한다.

확인하는 것 — RFC 4231 · RFC 5869 표준 벡터와 VM·네이티브 일치. **짓지 않은 것** — 상수 시간 보장, MAC 비교 op, 여러 블록 HKDF-Expand(L > 32), SHA-256 밖의 해시, 이어 먹이는 MAC. 함께 보기 — hash(SHA-256 자체) · aead (갈라낸 키를 쓰는 자리) · x25519 (그 비밀을 만드는 자리).

chacha — ChaCha20 스트림 암호

소스 lib/chacha.low
층 L0 — 순수 계산
권한 없음

키와 논스에서 **키스트림**을 만들어 입력과 XOR 한다(RFC 8439 §2.4). 대칭이라 같은 호출이 암호화이자 복호화다.

> **이것만으로는 안전하지 않다**
>
> ChaCha20 은 **감추기만** 한다 — 위조를 막지 못한다. 암호문 한 비트를 뒤집으면 평문의 같은 비트가 뒤집히고 아무도 눈치채지 못한다. 인증이 함께 있어야 쓸 수 있고 그 자리가 aead 다. 특별한 이유가 없으면 aead 를 쓴다. **논스 재사용은 치명적이다** — 같은 (키, 논스)로 두 번 흘리면 두 평문의 XOR 이 드러난다. 상수 시간을 약속하지 않고 감사받지 않았다.

32 비트 덧셈·XOR·회전만 쓰는 순수 계산이라 언어로 표현된다 — qround · rounds · keystream 이 그 글이고, 그것이 이 모듈이 말하는 **뜻의 정의**다. 다만 stream 은 낱말 chacha20 을 부른다: ChaCha 의 블록들은 서로 독립이라 넷·여덟을 한 레지스터에 나란히 실으면 그대로 넷·여덟 배가 되는데, 그 **폭**이 이 언어에 없다(실측 89 → 2439 MB/s). 답은 같고, 회귀 시험이 낱말과 위의 정의를 같은 입력으로 맞댄다. **32 비트 산술을 u64 위에서 한다.** 이 언어의 add 는 넘치면 멈추므로, 감김이 필요한 자리는 마스크를 **명시**한다(bit_and … 4294967295). 감김을 원하면 이름으로 말하라는 규율이 암호 구현에서 그대로 값을 낸다 — 조용한 감김은 조용한 결함이다.

op	하는 일	요구
prep	키·논스·카운터로 상태 16 워드를 세운다	key ≥ 32 · nonce ≥ 12
rounds	20 라운드(더블 라운드 10)를 돌린다	work 는 16 워드
keystream	상태에서 64 바이트 블록을 낸다	ks ≥ 64
stream	입력 전체를 흘려 out 에 쓴다	out ≥ len(inp)

표 50.1 — chacha 의 op

보통 쓰는 얼굴은 stream 이다. 답은 처리한 바이트 수이고 **0 이면 실패**다.

확인하는 것 — RFC 8439 §2.3.2 블록 함수 벡터와 §2.4.2 암호화 벡터. 블록 함수를 따로 재는 까닭은, 흐름만 맞추면 상태 배치가 틀려도 우연히 맞을 수 있고 그러면 다음 사람이 그 위에 지을 때 틀리기 때문이다. **짓지 않은 것** — 인증(aead), XChaCha20(확장 논스), 카운터 소진 감지, 상수 시간 보장.

poly — Poly1305 한 번 쓰는 인증자

소스 lib/poly.low
층 L0 — 순수 계산
권한 없음

메시지를 16 바이트 블록으로 먹여 **16 바이트 태그**를 낸다. “이 바이트열을 이 키를 아는 누군가가 보냈다” 를 증언한다(RFC 8439 §2.5).

> **키는 메시지마다 새것이어야 한다**
>
> Poly1305 는 **한 번 쓰는** 인증자다. 같은 키로 두 메시지를 인증하면 키를 복원할 수 있고, 그때부터 위조할 수 있다. 그래서 aead 는 메시지마다 논스로 1 회용 키를 만들어 넘긴다(key_gen). 이 모듈을 직접 쓴다면 그 규율을 직접 진다. 상수 시간을 약속하지 않고(경계 검사와 멈춤이 있다) 감사받지 않았다. 태그 비교는 부르는 쪽의 몫이다 — 이른 반환 없이 비교한다(aead.unseal 이 하는 방식).

130 비트 산술을 **26 비트 다섯 조각**으로 나눠 u64 위에서 한다. 곱의 최대가 2^{52} 언저리라 u64 안에서 안전하다 — **자리 배치가 곧 안전 논증**이고, 그래서 소스 머리에 그 수가 적혀 있다.

block 은 이 모듈이 말하는 **뜻의 정의**로 남는다. 다만 되풀이를 도는 자리는 aead 가 낱말 poly1305 로 부른다 — 상태의 자리가 같으므로(h 다섯 · r 다섯) setup 과 emit 은 그대로 언어에 있고, 가운데만 내려갔다(실측 331 → 1840 MB/s). 회귀 시험이 낱말과 block 의 되풀이를 같은 상태로 맞댄다.

op	하는 일
setup	키에서 상태(h 다섯 · r 다섯)를 세운다
block	16 바이트 블록 하나를 먹인다. addhi 는 꽉 찬 블록이면 1
emit	상태와 키로 태그 16 바이트를 낸다

표 50.1 — poly 의 op

addhi 가 인자인 까닭은 마지막 블록이 짧으면 최상위 1 비트를 다르게 놓아야 하기 때문이다. aead 는 pad16 으로 **모든 블록을 꽉 채우므로** 늘 1 을 넘긴다 — 패딩의 목적이 경우를 하나로 줄이는 것이라는 말이 여기서 실물이 된다.

확인하는 것 — RFC 8439 §2.5.2 태그 벡터와 VM·네이티브 일치. **짓지 않은 것** — 상수 시간 보장, 태그 비교, 키 재사용 감지, 이어 먹이는 상태의 직렬화.

aead — ChaCha20-Poly1305 봉인과 개봉

소스 lib/aead.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

메시지를 **감추고**(chacha) 그 메시지와 부가 데이터를 **한꺼번에 증언한다**(poly). TLS 1.3 의 기본 스위트(TLS_CHACHA20_POLY1305_SHA256)가 쓰는 바로 그것이다(RFC 8439 §2.8). **암호를 쓴다면 봉인은 여기서 시작한다.**

> **이 구현이 약속하지 못하는 것**
>
> **상수 시간이 아니다.** 이 코드는 비밀에 따라 갈라지지 않지만, 이 언어에는 경계 검사와 멈춤이 있어 타이밍을 약속할 수 없다. 원격 공격자가 시간을 잴 수 있는 자리에 그대로 쓰지 않는다. **감사받지 않았다** — 확인은 RFC 8439 §2.8.2 벡터이고 "표준이 정한 값이 나온다" 까지가 보증이다. **논스를 재사용하면 끝난다** — 같은 키로 같은 논스를 두 번 쓰면 스트림이 겹쳐 평문이 드러나고 태그 위조가 가능해진다. 이 모듈은 그것을 막지 않는다 — 세는 일은 부르는 쪽의 몫이다.

권한을 받지 않는다 — 열쇠와 논스를 **받아서** 일한다. 봉인과 개봉은 스스로 할당하지 않으므로 인자가 많다. 결과 자리(ct·msg·tag)와 작업 공간(otk·st·work· ks·pst·pad)을 모두 넘긴다.

op	하는 일	답
seal	msg 를 ct 로 봉인하고 16 바이트 tag 를 낸다	태그 길이 16. **0 이면 실패**
unseal	tag 를 먼저 확인하고 **맞을 때만** msg 를 낸다	평문 길이. **0 이면 실패(또는 위조)**
key_gen	논스마다 1 회용 Poly1305 키를 만든다	32

표 50.1 — aead 의 op

- seal 은 카운터 **1** 부터 흐름을 만든다 — 카운터 0 은 Poly1305 키로 이미 썼다. 그 한 칸이 겹치면 태그 키가 평문 스트림과 같아진다.
- MAC 이 먹는 배열은 AAD ‖ pad16(AAD) ‖ CT ‖ pad16(CT) ‖ le64(¦AAD¦) ‖ le64(¦CT¦) 다. pad16 덕에 모든 블록이 꽉 찬 16 바이트이고, 내부 호출이 모두 같은 모양이다.
- **실패는 0 이고, 그것이 위조와 같은 값이다.** unseal 의 0 은 "태그가 맞지 않다" 이거나 "버퍼가 모자라다" 다. 둘을 가르지 않는 것은 부르는 쪽이 둘 다 똑같이 다뤄야 하기 때문이다 — 어느 쪽이든 평문을 쓰면 안 된다. 0 을 받고 계속 가는 코드는 이미 틀렸다.
- 태그 비교는 **XOR 누적**이다 — 이른 반환이 없다. 이 언어에서 할 수 있는 최선이고, 위 경고가 말하는 한계이기도 하다.

확인하는 것 — RFC 8439 §2.8.2 벡터와 **변조 거절**(암호문이나 AAD 를 한 비트 바꾸면 unseal 이 0 을 답해야 한다), VM·네이티브 일치. **짓지 않은 것** — 상수 시간 보장, 이어 봉인하기, 논스 관리와 카운터 소진 감지, XChaCha20, 키 유도(hmac 의 HKDF). 함께 보기 — x25519 (키 합의).

x25519 — 곡선 위의 키 합의

소스 `lib/x25519.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

내 비밀 스칼라와 상대의 공개점으로 **둘만 아는 32 바이트**를 만든다(RFC 7748). 그 바이트열을 그대로 키로 쓰지 말고 `hmac` 의 HKDF 로 갈라 쓰는 것이 표준 관행이다.

> **이 구현이 약속하지 못하는 것**
>
> **상수 시간을 약속하지 않는다.** 사다리(ladder)는 비밀 비트로 분기하지 않도록 조건부 교환(`fsel`)으로 짰지만, 이 언어에는 경계 검사와 멈춤이 있어 타이밍을 약속할 수 없다. **감사받지 않았다. 작은 위수(small-order) 점을 거르지 않는다** — 상대가 보낸 공개키가 0 을 만드는 점일 수 있으니, 합의 결과가 모두 0 인지 확인하는 것은 부르는 쪽의 몫이다. 개인키를 만드는 일(엔트로피)은 여기 없다 — `random` 과 `cap random` 의 자리다.

```
rem w 는 u64 175 원소 이상 --- 작업 공간을 하나로 받는다
let n u64 be x25519.agree shared mysecret theirpub zbuf w .
guard eq n 32 . else return 1 .
```

왜 작업 공간이 하나인가. `scalarmult` 는 체 원소 아홉과 곱셈 자리를 따로 받는다 — 안에서 읽기에는 그 편이 낫다. 그런데 부르는 쪽에서 열넷을 늘어놓으면 파라미터 상한 16 을 금방 먹는다. 이 라이브러리를 쓴 첫 프로그램이 실제로 `E-IR-ARITY` 로 거절당했다. 그래서 `agree` 가 `w` 하나(175 원소 이상)를 잘라 쓰고, 자리 배치(체 원소 아홉 = `w[0..144]`, 곱셈 자리 = `w[144..175]`)는 소스 한 곳에만 있다.

표현을 고른 이야기. 소수는 `p = 2^255 - 19`. 흔한 참조 구현(ref10)은 조각을 26·25 비트로 담고 조각이 **음수가 될 수 있게** 두며 자리올림을 산술 오른쪽 시프트로 한다. 이 언어의 `shr` 은 **논리 시프트**라 그 배치를 쓸 수 없다. 그래서 **음수가 아예 없는 표현**(16 조각 × 16 비트, TweetNaCl 배치)을 골랐다. 뺄셈은 `a - b` 대신 `a + 4p - b` 로 한다 — `4p ≡ 0 (mod p)` 이므로 값은 같고 모든 중간값이 음이 아니다. 도구의 한계가 표현을 골랐는데, 고른 쪽이 추론하기 더 쉬운 쪽이었다.

op	하는 일	요구 · 답
`agree`	스칼라 × 점 → 공유 비밀 32 바이트(작업 공간 하나)	`w ≥ 175` · 나머지 32 바이트씩. 답 32, **0 이면 실패**
`scalarmult`	같은 계산, 작업 공간을 낱개로	인자 열넷
`fzero` · `fone` · `fcopy` · `fcar`	체 원소 초기화 · 복사 · 자리올림	위 배치를 아는 코드용
`fadd` · `fsub` · `fmul` · `fsq` · `fmul121665`	체 덧셈 · 뺄셈 · 곱 · 제곱 · 상수곱	〃
`finv` · `fsel` · `funpack` · `fpack`	역원 · 조건부 교환 · 바이트 ↔ 원소	〃

표 50.1 — `x25519` 의 op

체 산술이 `export` 인 것은 **시험하기 위해서**다(항등식을 잰다). 새 프로토콜을 짓는 것이 아니라면 `agree` 만 쓰면 된다. 같은 체를 `ed25519` 가 재사용한다.

확인하는 것 — RFC 7748 §5.2 스칼라곱 벡터, §6.1 DH 왕복(둘이 만나 같은 비밀을 얻는가), 체 산술의 항등식(`a·a⁻¹ = 1` 등), VM·네이티브 일치. **짓지 않은 것** — 상수 시간 보장, 작은 위수 점 거르기, 키 생성·클램핑 편의 op, X448.

aes — AES-128 블록 암호

소스 `lib/aes.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

16 바이트를 키로 뒤섞어 16 바이트를 낸다(FIPS 197). 그게 전부다 — 길이가 변하지 않고, 인증도 없고, 같은 입력은 언제나 같은 출력을 낸다. TLS 1.3 이 `TLS_AES_128_GCM_SHA256` 을 MUST 로 요구하므로(RFC 8446 §9.1) 있다.

혼자 쓰는 모듈이 아니다

> 같은 평문 블록이 같은 암호문 블록이 되므로 패턴이 그대로 보이고, 상대가 암호문을 고쳐도 알 수 없다. 실제로 필요한 것은 `gcm` 의 `encrypt · decrypt` 이고, 이 모듈은 그 밑의 부품이다. 상수 시간을 약속하지 않고 감사받지 않았다. AES-NI 를 쓰지 않는 순수 Lowent 구현이라 느리다 — 처리량이 필요하면 `aead` (ChaCha20-Poly1305)를 먼저 고려한다. 키 길이는 128 비트뿐이다.

op	하는 일	요구
`gmul`	GF(2^8) 곱(기약다항식 `0x11b`)	—
`ginv`	GF(2^8) 역원 — `x^254`(페르마)	`0` 의 역원은 `0`
`sbox · inv_sbox`	S-box 한 바이트 — 표가 아니라 **정의로 계산** · 역 S-box	—
`expand_key`	16 바이트 키 → 라운드 키 176 바이트	`rk ≥ 176 · key ≥ 16`
`encrypt_block`	`st` 16 바이트를 제자리에서 암호화	`rk ≥ 176 · st ≥ 16 · tmp ≥ 16`

표 50.1 — `aes` 의 op

복호 블록은 없다. GCM 은 카운터 모드라 암호화만 쓴다 — 복호도 암호화로 한다. `inv_sbox` 가 있는 것은 언젠가 지을 자리를 비워 둔 것이지 지었다는 뜻이 아니다.

S-box 를 표로 적지 않는다. 256 개 상수를 손으로 옮겨 적는 대신 정의 `S(x) = 아핀변환(x⁻¹ in GF(2^8))` 을 그대로 쓴다. 옮겨 적을 표가 없으면 옮겨 적다 틀릴 일도 없다. 시험이 계산한 256 개를 정본 표와 대조한다.

성능 — 재 보니 이랬다. 개발 저장소의 측정에서 `sbox` 한 번이 180 ns(표를 읽는다면 7.5 ns), `encrypt_block` 한 블록이 약 31 µs 이고 그중 93 % 가 `sbox` 160 회였다. AES-128-GCM 전체는 0.46 MB/s, ChaCha20-Poly1305 는 41.8 MB/s. 그중 15 배는 이미 갚았다 — `ginv` 가 `x^254` 를 254 번 곱하던 것을 제곱-곱으로 바꿔 곱셈이 254 → 15 가 됐고 시험 벡터는 한 자리도 움직이지 않았다. 표를 쓰면 GCM 이 2.5 MB/s 쯤 되겠지만 여전히 ChaCha20 보다 17 배 느리고, 표는 **비밀 색인 캐시 부채널** 을 들여온다 — AES 의 고전적인 공격 자리다. 느린 대비책을 위해 부채널을 사지 않는다.

확인하는 것 — FIPS 197 부록 B · C 시험 벡터, RFC 8448 §3 의 레코드 암호문(끝까지 써 본다), VM·네이티브 일치.

gcm — AES-128-GCM 인증 암호

소스 lib/gcm.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

"아무도 못 읽는다" 와 "아무도 못 고친다" 를 **한 번에** 준다(NIST SP 800-38D). aes 는 앞의 것만 준다.

무엇을 약속하고 무엇을 하지 않나

상수 시간을 약속하지 않고 감사받지 않았다. **논스를 절대 되풀이하지 않는다** — 같은 키로 같은 논스를 두 번 쓰면 GCM 은 **평문과 인증키를 함께 잃는다**. 이 모드에서 가장 비싼 실수다. **느리다** — 개발 저장소의 측정에서 초당 0.46 MB 로, 같은 기계의 ChaCha20-Poly1305(41.8 MB)보다 훨씬 느렸다. 그래서 스위트 선호는 ChaCha20 이다. 이 모듈은 규격이 MUST 라 **정확하게** 있는 것이지 빠르라고 있는 것이 아니다. decrypt 의 태그 검사 실패는 값이다 — 반환값을 반드시 본다.

AAD 는 무엇인가. 숨기지는 않지만 **고쳐지면 안 되는** 바이트다. TLS 1.3 에서는 레코드의 5 바이트 머리가 그것이다 — 길이는 관찰자가 어차피 보지만, 상대가 그것을 바꾸면 태그가 맞지 않아야 한다. aead 의 seal·unseal 과 달리 이 모듈은 encrypt·decrypt 라는 이름을 쓴다. 두 모듈이 같은 이름을 가지면 한정해 불러도 처리기가 다른 쪽 시그니처로 인자 수를 재는 결함이 있어 이름을 갈라 피했다.

op	하는 일	요구
gmul128	GHASH 의 GF(2^{128}) 곱 — z ← z·h	z ≥ 16 · h ≥ 16 · v ≥ 16
encrypt	msg → 암호문과 16 바이트 태그	nonce = 12 · key = 16
decrypt	태그를 **먼저 확인**하고 평문을 낸다	실패하면 0 을 답하고 평문을 주지 않는다

표 50.1 — gcm 의 op

뒷받침(scratch)은 호출자가 든다. 모자라면 0 을 답하고 아무것도 쓰지 않는다 — 가드가 지킨다.

GHASH 의 비트 차례 — 사람이 미끄러지는 자리. GF(2^{128}) 의 축약 다항식은 x^128 + x^7 + x^2 + x + 1 인데, GCM 은 **비트 차례를 뒤집어** 쓴다 — 블록의 첫 바이트 최상위 비트가 x^0 이다. 그래서 곱셈이 오른쪽 시프트로 돌고, 넘친 비트는 앞 바이트에 0xE1 로 되돌아온다. 이 관례를 반대로 읽으면 암호문은 맞는데 태그만 통째로 달라져 "복호가 안 된다" 로만 보인다.

논스에 대해 다시. GCM 은 카운터 모드라 같은 (키, 논스)가 같은 키스트림을 낸다. 두 평문의 XOR 이 드러나고, 더 나쁘게는 GHASH 인증키를 풀 수 있는 방정식이 생겨 상대가 임의의 메시지를 위조할 수 있다. TLS 1.3 은 논스를 **시퀀스 번호에서 유도**해 이것을 피한다(tls13 의 record_nonce). 직접 쓴다면 그 방식을 따른다 — 난수 논스는 96 비트에서 충돌이 생각보다 이르다.

확인하는 것 — NIST GCM 시험 벡터, RFC 8448 §3 의 레코드 암호문(암호문과 태그까지 바이트 그대로), VM·네이티브 일치.

`crypto_hw` — 기계가 도와주는 암호 셈

소스 `lib/crypto_hw.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

암호가 쓰는 셈 가운데 **CPU 에 명령이 있는 것**을 모은다. 오늘은 GF(2^{128}) 곱 하나다 — GCM 의 GHASH 가 그 위에 서고, CRC 도 같은 셈이다.

> **이 모듈에 `unsafe` 가 없다**
>
> 기계 명령을 내는 것은 **처리기**이고, 이 소스가 쓰는 것은 평범한 낱말(`clmul_lo` · `clmul_hi`)이다. 그래서 효과가 번지지 않는다 — 부르는 쪽은 `effects none` 그대로다. 어셈블리를 직접 쓰면 `unsafe` 가 TLS 까지 올라간다(30장 장의 「흡수 경계」).

op	하는 일	요구
`clmul128`	캐리 없는 곱 한 번 — 128 비트 결과를 낱말 둘로	`out ≥ 2`
`gf128_mul`	GF(2^{128}) 곱 — GCM 의 뒤집힌 표현, 카라추바로 곱셈 셋	`out · x · h` 각 `≥ 2`

표 50.1 — `crypto_hw` 의 op

캐리 없는 곱셈이란. 자리 올림 없이 하는 곱셈이다 — 더하기가 배타합(XOR)이다. 그래서 답이 입력 비트의 조합이고, 암호와 검사합이 그 위에 선다. 답이 128 비트인데 이 언어에는 128 비트 타입이 없으므로 **낱말 둘**(아랫말 · 윗말)로 나온다.

빠르기는 빌드가 정한다. `lowentc --hw auto` 로 지으면 기계에 명령이 있을 때 그것을 쓰고, 없으면 같은 답을 내는 셈으로 돈다. `--hw none`(기본)이면 언제나 셈이다. **답은 어느 쪽이든 같다** — 다른 것은 빠르기와 타이밍 성질뿐이다.

재 보니 이랬다. 잎 `ghash` 82 → 6562 MB/s · 잎 `aes_ctr` 80 → 5475 · AES-128-GCM 전체 41 → 2848 MB/s(4 MiB, gcc, 정렬 판 셋 가운데 각 열의 최선). AES 쪽 명령(`--hw aes`)과 함께 켜야 그 수가 나온다 — 하나만 켜면 병목이 다른 쪽으로 옮겨 갈 뿐이다.

명령을 담는 것으로 끝이 아니었다. 같은 명령을 쓰면서 처음에는 천장의 절반만 뽑아냈다. 남은 절반은 **쓰는 법**이었다 — 덩이를 여덟씩 엮고, 축약을 여덟 블록에 한 번만 하고, 카운터를 메모리 왕복 없이 레지스터에서 세우고, 축약 자체를 곱셈 둘로 줄였다. 잎 `aes_ctr` 는 이제 같은 상자의 OpenSSL 과 **같은 속도**다.

확인하는 것 — 이 모듈의 `gf128_mul` 과 처리기의 잎 `ghash` 가 **같은 답**을 내는지 회귀 시험이 본다. 같은 곱을 서로 다른 자리에서 셈하므로, 한쪽이 틀리면 다른 쪽이 그것을 드러낸다. 기계 명령 판과 셈 판, 그리고 VM 까지 셋을 같은 입력으로 맞대는 것도 같은 자리에서 한다.

> **상수시간을 약속하지 않는다**
>
> 기계 명령 경로는 표를 읽지 않아 타이밍 성질이 더 낫다. 그래도 이 저장소의 크립토는 상수시간을 약속하지 않고 감사받지 않았다 — 다른 자리가 남아 있기 때문이다. 약속하지 않는다는 말을 지키려고 이 사실을 적는다.

`bigint` — 큰수 모듈러 산술

소스 `lib/bigint.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

`u64` 하나에 들어가지 않는 수를 **모듈러 곱하고 거듭제곱한다**. RSA 의 2048 비트 수 같은 것이다.

무엇을 약속하고 무엇을 하지 않나

상수 시간을 약속하지 않고 감사받지 않았다. **필요한 만큼만 지었다** — RSA 검증(`s^e mod n`)과 곡선 산술에 드는 것뿐이다. 큰수 나눗셈 · 음수 · GCD · 역원은 없다. 지수 `e` 는 **`u64` 하나**다 — RSA 의 공개 지수(보통 65537)를 담으려고 그렇게 좁혔다. 안 지은 것은 틀릴 수도 없다.

표현 — 조각 폭은 누산이 정한다. 수는 **16 비트 조각(limb)** 을 `u64` 에 담아 리틀엔디언 차례로 늘어놓는다. 곱이 16×16 = 32 비트이고 128 조각을 더해도 2^{39} 라 `u64` 안에서 안전하다. 32 비트 조각을 쓰면 곱이 2^{64} 에 닿아 누산이 넘친다. "레지스터가 64 비트니 조각도 64 비트" 는 자연스러워 보이지만 틀린다 — 조각 폭을 정하는 것은 곱이 아니라 누산이다.

op	하는 일	요구 · 비고
`zero`	`a` 의 앞 `k` 조각을 0 으로	`a ≥ k`
`from_bytes` · `to_bytes`	**빅엔디언** 바이트 ↔ 조각	RSA 의 인코딩
`from_bytes_le` · `to_bytes_le`	**리틀엔디언** 바이트 ↔ 조각	곡선 쪽 관례
`ge_mod`	`a ≥ n` 인가(0 · 1)	`k` 조각 비교
`dbl_mod`	`a ← 2a mod n`	R^2 를 만드는 데 쓴다
`mont_mul`	`out ← a·b·R⁻¹ mod n` (CIOS)	`t` 는 작업 공간
`n0inv16`	`−n⁻¹ mod 2^16` — 몽고메리 상수	`n` 의 맨 아래 조각만 받는다
`mod_exp`	`out ← bse^e mod n`	`e` 는 `u64` · `r2`·`acc`·`tmp`·`t` 는 작업 공간

표 50.1 — `bigint` 의 op

왜 몽고메리인가 — 나눗셈을 짓지 않으려고. 보통의 모듈러 곱 `a·b mod n` 에서 `mod` 는 큰수 나눗셈이다. 몽고메리 곱은 대신 `a·b·R⁻¹ mod n` 을 내는데 `R = 2^(16k)` 이라 나누기가 **시프트**가 된다. 대가로 값을 몽고메리 영역으로 옮겼다 되돌려야 하고 $R^2 \bmod n$ 이 필요한데, 그것도 나눗셈 없이 얻는다 — 1 을 `2·16k` 번 배로 늘리면서(`dbl_mod`) 넘칠 때마다 `n` 을 뺀다. 이 모듈의 설계 전체가 이 한 줄이다: **나눗셈을 짓지 않으려고 곱을 바꿨다.**

확인하는 것 — 파이썬 정수 산술과의 바이트 대조, RSA-PSS 검증이 끝까지 통하는지(`rsa`), VM·네이티브 일치. **짓지 않은 것** — 상수 시간, 큰수 나눗셈, 음수, GCD, 모듈러 역원(곡선 쪽 역원은 `p256` 이 페르마 거듭제곱으로 한다), 아주 큰 `e`.

rsa — RSASSA-PSS 검증

소스 lib/rsa.low
층 L0 — 순수 계산
권한 없음

상대가 보낸 RSA-PSS 서명이 그 공개키와 메시지에 맞는지 답한다(RFC 8017 §8.1.2). TLS 1.3 의 rsa_pss_rsae_sha256 이 이것이다.

> **무엇을 약속하고 무엇을 하지 않나**
>
> 상수 시간을 약속하지 않고 감사받지 않았다. **PSS 만** 한다 — PKCS#1 v1.5 는 짓지 않는다. RFC 8446 §4.4.3 이 CertificateVerify 에서 금지하기 때문이다. SHA-1 계열, 키 생성, 복호(RSA 암호화)는 없다. 서명 생성도 없다 — TLS 1.3 서버가 언젠가 필요로 할 수 있지만, 먼저 선 것은 키와 서명이 작은 p256 이다.

옛 PKCS#1 v1.5 는 패딩이 결정적이고 구조가 얕아 여러 공격이 붙었다. PSS 는 솔트를 섞고 마스크로 덮는다. PSS 검증은 세 조각이다.

```
① s^e mod n  →  EM (인코딩된 메시지, emLen = 키 길이)
② MGF1(SHA-256) 로 마스크를 만들어 DB 를 벗긴다
③ H' = SHA-256(0x00×8 ‖ mHash ‖ salt) 가 EM 안의 H 와 같은가
```

①은 bigint 의 mod_exp 이고, ②③이 이 모듈의 몫이다. 0x00×8 여덟 바이트는 장식이 아니다 — 서명 대상과 인증서 서명을 가르려고 규격이 넣은 접두사다. 빼먹으면 다른 문맥의 서명을 재활용할 수 있게 된다.

op	하는 일	요구
mgf1	MGF1-SHA256 마스크 생성 함수	out ≥ n · w 는 뒷받침
pss_verify	EM 이 주어졌을 때 PSS 인코딩이 맞는가	②③만 한다
verify_pss	서명 · 모듈러스 · 지수 · 해시 → 0 또는 1	①②③ 전부

표 50.1 — rsa 의 op

보통은 verify_pss 하나만 쓴다. pss_verify 가 따로 나와 있는 것은 ①과 ②③을 갈라서 시험하려는 것이다.

확인하는 것 — 표준 PSS 시험 벡터(양성), 음성 둘(서명 1 비트 · 해시 1 비트), MGF1 을 참조 구현과 대조, VM·네이티브 일치. 음성이 둘인 까닭은 서명을 바꾼 것과 메시지를 바꾼 것이 다른 실패이기 때문이다 — 하나만 보면 나머지 하나가 통과하는 구현을 거르지 못한다.

p256 — NIST P-256 곡선과 ECDSA 검증

소스 `lib/p256.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

타원곡선 `y² = x³ - 3x + b` 위의 점 산술과, 그 위에 선 **ECDSA 서명 검증**이다. TLS 1.3 의 `ecdsa_secp256r1_sha256` 이 이 곡선을 쓴다.

> **무엇을 약속하고 무엇을 하지 않나**
>
> 감사받지 않았다. 검증용 `smul` 은 스칼라 비트로 **분기한다** — 검증에서 그 스칼라는 공개 정보라 괜찮다. 비밀 스칼라(개인키 · 논스)에는 `smul_ct` 를 쓴다 — 걸음마다 두 배와 덧셈을 둘 다 하고 쓸지 말지를 분기가 아니라 마스크로 고른다. 그 약속은 **비밀에 의존하는 분기와 메모리 접근이 없다** 는 데까지이고, 캐시 계층 · 전력 · 전자기는 약속하지 않는다. 점 압축 해제와 P-384 · P-521 은 없다.

왜 이 곡선인가. 공개 CA 가 실제로 발급하고 브라우저가 실제로 받아들이는 것이 P-256 과 RSA 다. `ed25519` 가 수학적으로 더 깔끔하지만 공개 웹에서는 통하지 않는다 — 수학이 아니라 생태계가 정한다. 서명 **생성**은 `ecdsa` 가 이 모듈 위에서 한다.

op	하는 일
`limbs`	조각 수(16 = 256 비트)
`msub` · `madd`	모듈러 뺄셈 · 덧셈
`minv`	모듈러 역원(페르마 거듭제곱)
`pmul`	몽고메리 곱(`bigint` 의 것을 재사용)
`make_r2`	`R² mod p` — 몽고메리 영역으로 들어가는 상수
`pdbl` · `padd`	야코비 좌표 점 두 배 · 점 덧셈
`is_zero`	무한원점인가
`smul`	스칼라 곱 `[k]P` — 공개 스칼라용
`smul_ct`	스칼라 곱 — 비밀 스칼라용, 분기 없이 마스크로 고른다
`ecdsa_verify`	서명 `(r, s)` 가 해시 `e` 와 공개키에 맞는가

표 50.1 — `p256` 의 op

곡선 상수(p · n · Gx · Gy)는 호출자가 넘긴다.

왜 야코비 좌표인가 — 역원을 미루려고. 아핀 좌표의 점 덧셈은 매번 모듈러 역원을 요구하고, 역원은 곱보다 수백 배 비싸다. 야코비 좌표는 분모를 좌표 안에 들고 다녀 역원을 맨 끝 한 번으로 미룬다. 같은 이유로 `ed25519` 는 확장 좌표를 쓰는데, 그쪽 덧셈 공식은 **완비**라 예외 갈래가 없고 여기는 **있다**(같은 점이면 다른 식). 그 갈래가 `padd` 안에 있다.

확인하는 것 — NIST CAVP 의 P-256/SHA-256 서명 검증 벡터 양성 2 · **음성 8**, 파이썬 참조 곡선과의 체 · 스칼라곱 대조, VM·네이티브 일치. 서명 검증 시험에 음성이 없으면 "언제나 참" 을 걸러내지 못한다 — 양성만 있는 검증 시험은 검증하지 않는다.

p384 — NIST P-384 곡선과 ECDSA 검증

소스 lib/p384.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

곡선 y² = x³ − 3x + b(소수 p = 2^384 − 2^128 − 2^96 + 2^32 − 1) 위의 **ECDSA 서명 검증**이다. TLS 1.3 의 ecdsa_secp384r1_sha384 가 이 곡선을 쓴다. 공개 웹의 여러 중간 인증기관과 그 뿌리가 P-384 로 서명하므로, 인증서 체인을 끝까지 확인하려면 이 곡선이 있어야 한다.

무엇을 약속하고 무엇을 하지 않나

감사받지 않았다. 내보내는 것은 **검증** 하나뿐이고, 서명 생성 · 점 압축 해제 · 상수시간 스칼라 곱은 없다. 검증에서 쓰는 스칼라는 공개 정보라 분기해도 괜찮다 — 비밀 스칼라를 다루는 일은 이 모듈이 하지 않는다.

왜 p256 을 고쳐 쓰지 않고 따로 두나. p256 은 조각 수 16 을 루프 상한과 작업 공간 자리마다 박아 두었고, 검산이 끝난 채 여러 모듈이 쓰고 있다. 그것을 조각 수로 매개변수화하면 검산이 끝난 코드를 건드리게 된다. 그래서 곡선마다 제 모듈을 두는 규율을 따랐다 — 이 파일은 p256 을 16 조각에서 24 조각으로 옮긴 것이고 **식은 한 글자도 다르지 않다**. 다른 것은 자리 수뿐이다.

자리	p256	p384
수 하나(16비트 조각)	16	24
점 하나(야코비 좌표 X · Y · Z)	48	72
몽고메리 임시	18	26
지수 비트	256	384

표 50.1 — p256 과 p384 는 자리 수만 다르다

op	하는 일
ecdsa_ok	서명 (r, s) 가 해시 e 와 공개키 pub 에 맞는가 — ok 1 맞다 · ok 0 아니다 · error short_workspace 작업 공간이 모자라다

표 50.2 — p384 의 op

ecdsa_ok 는 곡선 상수(p · n · Gx · Gy)를 호출자에게서 받고, 공개키는 아핀 좌표 x ‖ y(48 조각), 작업 공간 w 는 1364 조각 이상이어야 한다(p256 의 924 를 1.5 배 한 자리). **모자라면 실패로 말한다** — 예전에는 «자리가 모자람» 과 «서명이 틀림» 이 둘 다 0 이었고, 작업 공간을 잘못 준 프로그램이 «서명이 틀렸다» 로 보였다.

검증 계산은 이렇다. r 과 s 가 1 … n−1 안에 있는지를 **먼저** 보고, 그다음

```
w  = s⁻¹ mod n
u1 = e·w mod n ,  u2 = r·w mod n
R  = u1·G + u2·Q        (Q = 공개키 점)
R.x ≡ r (mod n) 이면 참
```

모듈러 곱은 bigint 의 몽고메리 곱을 그대로 쓴다 — p 와 n 이 둘 다 홀수라 조건이 맞고, 새 산술을 또 짓지 않는다. 좌표는 p256 과 같은 까닭으로 야코비 좌표다(역원을 맨 끝 한 번으로 미룬다).

확인하는 것 — 우리가 지은 것이 아닌 오라클 둘에 맞댄다. ① openssl(secp384r1 · SHA-384)이 짓고 openssl dgst -verify 가 맞다고 한 서명 한 벌 — 양성 하나와 **음성 셋**(s 한 비트 · 해시 한 비트 · s = 0). ②

진짜 인증기관이 서명한 진짜 인증서 체인(몇몇 공개 사이트의 중간 마디가 P-384 다). NIST CAVP 시험 벡터는 **아직 돌리지 않았다** — 그 파일이 저장소에 없다. 이 모듈의 소비자는 `verify` 다.

ecdsa — ECDSA P-256 서명 생성

소스 lib/ecdsa.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

개인키와 메시지 해시를 받아 서명 (r, s) 를 낸다(RFC 6979). 검증은 p256 의 ecdsa_verify 다. TLS 서버가 CertificateVerify 를 자기 개인키로 서명할 때 쓴다 — 서명 없이는 서버가 없다.

무엇을 약속하고 무엇을 하지 않나

상수 시간은 여기서만, 크기를 적어 약속한다. 없애는 것은 비밀 스칼라(개인키 d · 논스 k)에 의존하는 분기와 메모리 접근이다. 몽고메리 곱의 마지막 조건부 뺄셈 · 캐시 계층 · 전력 · 전자기 · 컴파일러가 만들어 낼 수 있는 것은 약속하지 않는다. 감사받지 않았다. P-256 뿐이고 키 생성은 없다. r = 0 또는 s = 0 이면 다시 하지 않고 실패로 답한다(확률 2^{-128} 쯤) — 그 갈래를 지으면 시험할 수 없는 코드가 생기고, 시험 안 된 암호 코드는 없느니만 못하다.

왜 논스를 난수가 아니라 유도하는가. ECDSA 에서 논스는 한 번 새거나 한 번 되풀이되면 개인키가 통째로 드러난다.

```
s = k⁻¹ (h + r·d)   ⇒   d = (s·k − h) / r
```

k 를 알면 그 한 줄로 끝이고, 같은 k 를 두 번 쓰면 두 서명에서 k 가 풀린다. 그리고 난수원이 실패하는 경로는 조용하다 — 빈 엔트로피, 포크 뒤의 같은 상태, 가상 머신 스냅숏 복원. 그래서 **난수원을 아예 쓰지 않는다.** k = HMAC-DRBG(개인키, 메시지 해시) 다. 같은 (키, 메시지)면 같은 서명이 나오고, 그것은 결함이 아니라 성질이다.

op	하는 일	요구
nonce6979	RFC 6979 §3.2 — (d, h) → k	w ≥ 480 바이트 · wu ≥ 16 조각
sign	(d, h) → r ‖ s 64 바이트	wb ≥ 512 바이트 · wu ≥ 908 조각

표 50.1 — ecdsa 의 op

곡선 상수(p · n · Gx · Gy)는 호출자가 준다 — 이 모듈은 표를 들고 있지 않다. 서명은 p256 의 smul_ct 만 쓰고, 그것을 시험이 확인한다. k^{-1} 은 페르마 거듭제곱인데 지수(n − 2)가 공개라 연산 순서가 고정이다.

넉넉함은 공짜가 아니다. 작업 공간을 처음엔 1052 조각으로 잡았는데 그러면 --run 의 배열 인자 상한(u64 1024)을 넘어 VM 에서 시험할 수 없었다. 빈 자리를 두 번 당겨 908 로 줄였다 — 시험할 수 없는 크기는 크기가 아니라 결함이다. **성능** — sign 은 파라미터 · 지역 · 명령이 많아 타입이 붙은 빠른 경로에 들어가지 않는다. 연결당 서명 한 번이라 지금은 견딜 만하다.

확인하는 것 — RFC 6979 §A.2.5 공식 벡터(논스 k, 메시지 "sample" · "test" 의 r · s 를 바이트로), 낸 서명을 우리 검증기가 받는지, 한 비트를 뒤집으면 거절하는지. 벡터 대조는 "규격대로인가", 왕복은 "우리 두 쪽이 서로 맞는가" 를 묻는다.

ed25519 — Ed25519 서명 검증

소스 lib/ed25519.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

Curve25519 와 **같은 소수** `p = 2^255 - 19` 위의 비틀린 에드워즈 곡선에서 서명을 검증한다(RFC 8032 §5.1.7). 소수가 같으므로 `x25519` 의 체 산술을 그대로 재사용한다 — 같은 소수면 같은 체다. 그쪽은 키 합의, 여기는 서명 검증이다.

> **무엇을 약속하고 무엇을 하지 않나**
>
> 상수 시간을 약속하지 않고 감사받지 않았다. **서명 생성은 없다.** `ed25519ph` · `ed25519ctx` · Ed448 도 없다. **공개 웹 HTTPS 서버의 인증서에는 이 알고리즘을 쓸 수 없다** — 공개 CA 가 Ed25519 서버 인증서를 사실상 발급하지 않는다. 그쪽은 `p256` 이다. 이 모듈은 **상대의** 서명을 검증하는 데 쓴다.

에드워즈 덧셈은 완비(complete)다. 확장 좌표 `(X:Y:Z:T)`, `xy = T/Z` 를 쓴다. 에드워즈 덧셈 공식에는 예외가 없다 — 야코비와 달리 "같은 점이면 다른 식" 이 없다. `p256` 의 `padd` 안에 있는 갈래가 여기엔 아예 없고, 갈래가 없으면 그 갈래에서 틀릴 수도 타이밍으로 샐 수도 없다.

애매한 입력은 거절하는 쪽을 골랐다. RFC 8032 는 비정규 인코딩과 작은 위수 점의 처리를 구현마다 달라도 되게 둔다. 여기서는 거절한다 — `S < L` 을 강제하고 (안 하면 같은 메시지에 유효한 서명이 여럿 생긴다), 압축 해제가 실패하면 거절한다. 규격이 "둘 다 된다" 고 할 때 고른 쪽을 적어 두는 것이 문서의 일이다.

op	하는 일
`verify`	서명 `sig` 가 공개키 `pub` 와 메시지 `msg` 에 맞는가 — 보통은 이것만 쓴다
`make_d` · `make_l` · `make_base`	곡선 상수 `d` · 위수 `L` · 기저점 `B`
`eadd` · `esmul`	점 덧셈 · 스칼라 곱
`compress` · `decompress`	점 ↔ 32 바이트 인코딩
`fpow` · `feq` · `fbit0`	체 거듭제곱 · 같은가 · 최하위 비트
`pow2_minus` · `reduce_l`	`2^n - k` · `mod L` 축약

표 50.1 — `ed25519` 의 op

검증식은 `[S]B = R + [k]A`, `k = SHA-512(R ‖ A ‖ M) mod L` 이다.

옮겨 적을 수밖에 없는 상수는 성질로 검사한다. `d` 와 `√-1` 과 `B` 는 계산으로 낸다 — 적을 것이 없다. 위수 `L` 은 유도되지 않아 적어야 했다. 그래서 시험이 `[L]B` 가 항등원인지 본다. 틀리게 적으면 그 검사가 깨진다 — 옮겨 적은 것을 믿지 않고 재는 방법이다.

확인하는 것 — RFC 8032 §7.1 양성 3 · **음성 6**, `[L]B = 항등원`, VM·네이티브 일치.

der — DER 최소 파서

소스 lib/der.low
층 L0 — 순수 계산
권한 없음

X.509 인증서 바이트에서 **공개키가 어디서 시작해 몇 바이트인지** 알려 주고, PKCS#8 개인키 파일에서 EC 스칼라의 자리를 찾는다. 그 키로 서명을 검증하는 것은 p256 · rsa · ed25519 의 일이다.

> **PKI 가 아니다**
>
> 체인 검증 · CA 신뢰 저장소 · 이름 대조 · 유효기간 · 폐기 · 확장(Extensions) — **하나도 하지 않는다**. 그러므로 이것만으로 "이 인증서를 믿어도 되는가" 에 답할 수 없다. 부르는 쪽이 공개키를 **고정(pinning)** 하거나 자기 규칙으로 판단해야 한다. 반쯤 지은 PKI 는 없는 것보다 나쁘다 — 믿을 수 있다고 착각하게 만든다.

길이는 상대가 쓴 것이다. DER 의 모든 길이 필드는 공격자가 정한다. 그래서 이 파서는 셋을 지킨다 — 어떤 길이도 버퍼를 넘으면 즉시 실패한다(0 을 답한다). 긴 형식 길이는 4 바이트까지만 받는다. 재귀를 하지 않는다(깊이는 부르는 쪽이 정하는 반복이다). **모든 실패는 0 이다** — 예외가 아니라 값이다.

op	하는 일
tag_at	off 위치의 태그 바이트
value_off · value_len	그 TLV 의 **값**이 시작하는 자리 · 길이
next_off	다음 형제 TLV 의 자리
is_spki	이 자리가 SubjectPublicKeyInfo **모양**인가
find_spki	인증서 안에서 SPKI 를 찾는다
alg_oid_off · alg_oid_len	알고리즘 OID 의 자리 · 길이
key_off · key_len	공개키 비트열의 자리 · 길이
rsa_n_off · rsa_n_len · rsa_e	RSA 키 안의 모듈러스 · 지수
p8_alg_off · p8_inner_off · p8_inner_len	PKCS#8 의 알고리즘 · 안쪽 키의 자리 · 길이
ec_priv_off · ec_priv_len	EC 개인키(스칼라)의 자리 · 길이

표 50.1 — der 의 op

자리로 세지 않고 모양으로 찾는다. SPKI 는 인증서의 여섯째이거나 일곱째 필드다 — version 이 있느냐에 따라 다르다. 자리로 세면 옛 인증서에서 조용히 어긋난다. 그래서 is_spki 는 SEQUENCE { SEQUENCE { OID … }, BIT STRING } 이라는 모양을 본다. 세는 것은 앞이 바뀌면 틀리고, 알아보는 것은 틀리지 않는다.

확인하는 것 — 파싱 결과를 눈으로 보지 않는다. RFC 8448 §3 의 실제 인증서에서 키를 꺼내 그 핸드셰이크의 CertificateVerify 서명을 **검증한다**. 파싱이 한 바이트라도 밀리면 서명이 맞지 않는다. 음성 둘(서명 · 전사 각 1 비트)도 본다. "꺼낸 값이 그럴듯한가" 를 사람이 보는 검사는 검사가 아니다.

pem — PEM 봉투 벗기기

소스 lib/pem.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

-----BEGIN CERTIFICATE----- 와 -----END CERTIFICATE----- 사이의 접힌 base64 에서 **가운데 바이트(DER)** 를 꺼낸다(RFC 7468). 키 파일도 같은 모양이고 라벨만 다르다. codec 은 base64 를 알지만 줄 접기는 모른다 — 이 모듈이 머리말을 벗기고 접힘을 펴서 넘긴다. 인증서는 밖에서 받는다(certbot 같은 도구가 파일로 놓아 준다). ACME 는 짓지 않는다.

무엇을 약속하고 무엇을 하지 않나

암호화된 PEM 을 열지 못한다(Proc-Type: 4,ENCRYPTED). **첫 번째 것만** 낸다 — 한 파일에 여러 개(인증서 체인)가 있으면 뒤엣것은 부르는 쪽이 다시 부른다. URL-safe base64 는 없다. **파서이지 신뢰 판단이 아니다** — 검증은 하지 않는다.

op	하는 일
find_from	hay 안에서 needle 찾기(없으면 len hay)
body_off	-----BEGIN <라벨>----- **다음 줄**의 자리. 0 = 없음
end_off	-----END <라벨>----- 의 자리. 0 = 없음
unwrap	PEM 한 덩이 → DER 바이트. option u64(쓴 바이트 수), 실패는 none

표 50.1 — pem 의 op

unwrap src label scratch out 의 scratch 는 두 몫을 진다 — 머리말 조립(앞)과 펴 놓은 base64(뒤). len scratch ≥ len src + 라벨 길이 + 16 이면 넉넉하다.

라벨을 요구하는 이유. unwrap 은 무엇을 여는지 이름으로 받고, BEGIN 과 END 의 라벨이 다르면 거절한다. 한 파일에 여러 개가 들어 있는 것이 정상이다. 짝을 맞추지 않으면 CERTIFICATE 를 열려다 다음 것의 BEGIN 까지 삼켜 쓰레기를 디코드한다 — 그리고 base64 는 쓰레기도 조용히 받아들인다.

줄 끝은 LF 도 CRLF 도 받는다 — http 와 반대다. 그쪽은 경계가 곧 보안이라 맨 LF 를 거절한다(요청 밀반입). 여기는 파일 형식이고 실제 파일이 둘 다 쓴다. 엄격함은 미덕이 아니라 도구다 — 무엇을 막는지에 따라 세기가 달라진다.

왜 decode 가 아니라 unwrap 인가. utf8.decode 가 이미 있고, 이름이 겹치면 한정해 불러도 처리기가 다른 모듈의 시그니처로 타입을 재는 알려진 결함이 있다. 두 모듈을 함께 쓰는 시험이 그 조합이 깨지는 것을 잡았다. 라이브러리는 혼자 초록인 것으로 충분하지 않다 — 조합되어야 쓸 수 있다.

PEM → DER → PKCS#8 → 32 바이트 스칼라로 가는 길은 der 와 함께다 — pem.unwrap src "PRIVATE KEY" sc buf 로 DER 을 얻고, der.p8_inner_off 와 der.ec_priv_off 로 스칼라의 자리를 찾는다.

확인하는 것 — openssl 이 낸 산물과의 대조(인증서 DER 375 바이트가 같고, PKCS#8 안의 스칼라가 자리 36 · 길이 32), 라벨 불일치와 봉투 없음의 거절, VM·네이티브 일치. 시험 벡터의 스칼라는 합성값 01 02 … 20 이다 — 구조는 openssl 이 낸 그대로이므로 파서를 재는 힘은 같고, 명백히 비밀이 아니다.

x509 — X.509 인증서 읽기

소스 lib/x509.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

X.509 인증서(DER)의 **뼈대를 읽어 준다**. 서명이 덮는 부분은 어디서 어디까지인지, 발급자와 주체는 누구인지, 공개키 · 유효기간 · 확장은 어디 있는지를 **자리(오프셋)와 길이**로 낸다. der 가 «공개키만 꺼낸다» 에서 멈춘 자리의 다음 칸이다.

읽기만 한다

> 이 모듈은 인증서를 **확인하지 않는다**. 서명이 맞는지, 체인이 이어지는지, 이름이 맞는지는 verify 가 이 모듈 위에서 답한다. 짓지 않은 것: 폐기 확인(CRL · OCSP) · 이름 제약 · 정책 · v1/v2 의 옛 모양 · UTF-8 이 아닌 이름 비교.

인증서의 모양과 이 모듈의 op 이 가리키는 자리는 이렇다.

```
Certificate
├─ tbsCertificate ──────────────── tbs_off … tbs_end   ← 서명이 덮는 바이트
│   ├─ [0] version (없을 수도 있다)
│   ├─ serialNumber                tbs_field 0
│   ├─ signature                   tbs_field 1
│   ├─ issuer (발급자 이름)        issuer_off · elem_len
│   ├─ validity                    not_before · not_after
│   ├─ subject (주체 이름)         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	하는 일
tbs_off · tbs_end	서명이 덮는 바이트의 시작과 끝. 서명은 이 구간의 **원본 바이트** 위에서 확인한다
sigalg_oid_off · sigalg_oid_len	바깥 서명 알고리즘 OID 의 자리와 길이
sig_off · sig_len	서명 값(BIT STRING 의 «남은 비트 수» 바이트를 건너뛴 자리)
tbs_field	tbsCertificate 안의 n 번째 필드 자리(0 serial · 1 signature · 2 issuer · 3 validity · 4 subject · 5 공개키)
issuer_off · subject_off · elem_len	발급자 · 주체 이름 요소 **전체**의 자리와 길이 — 이름은 바이트로 견준다
spki_off	공개키(SubjectPublicKeyInfo)의 자리
not_before · not_after	유효기간을 견줄 수 있는 한 수(YYYYMMDDhhmmss)로. 시간대는 Z 만 받는다
ext_value_off	OID 가 2.5.29.<n> 인 확장의 값 자리(17 = 주체 대체 이름 · 19 = basicConstraints)
is_ca	남을 발급할 수 있는 인증서인가. 확장이 **없으면 거짓**이다
san_next	주체 대체 이름의 DNS 이름을 하나씩 준다(cur 가 0 이면 첫째, 0 을 돌려주면 끝)

표 50.1 — x509 의 op — 자리를 내는 op 은 0 이면 «성하지 않다»

값을 베끼지 않는다. 모든 op 이 자리와 길이만 낸다. 64 KiB 아레나에 인증서 몇 장을 한꺼번에 들어야 하고, 서명은 원본 바이트 위에서 확인해야 하기 때문이다 — 베낀 자리에서 확인하면 베끼기가 틀려도 서명이 맞는 것처럼 보일 수 있다.

길이는 전부 상대가 쓴 것이다. 인증서는 아직 아무것도 믿을 수 없는 바이트열이다. 자리를 내는 길은 모두 `der.value_off` · `der.value_len` 을 지나고, 그 둘이 버퍼 밖을 막는다. 이 모듈은 그 위에 **차례**를 얹는다 — X.509 의 필드에는 이름표가 없어서 몇 번째인지가 곧 무엇인지이고, 차례를 안 보면 남이 끼워 넣은 필드를 제 자리 것으로 읽는다.

`is_ca` 의 기본값이 중요하다. basicConstraints 확장이 없으면 CA 가 아니다(RFC 5280 §4.2.1.9). 이 기본값을 놓치면 잎 인증서가 중간 인증기관 노릇을 할 수 있다 — 체인 검증에서 가장 흔한 구멍이다.

`verify` — 인증서 서명과 체인의 한 마디 확인

소스 `lib/verify.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

`x509`가 인증서를 **읽어** 주면, 이 모듈은 그 위에서 **인증서 한 장이 다른 한 장에게 서명받았는가**를 답한다. 서명 확인에 더해 유효기간 · 이름 이어짐 · 호스트 이름 대조까지, 체인의 **한 마디**를 잇는 데 필요한 판단을 한 자리에 모았다.

무엇을 약속하고 무엇을 하지 않나

감사받지 않았다. 받는 서명 알고리즘은 넷뿐이다 — RSA v1.5 + SHA-256 · RSA v1.5 + SHA-384 · ECDSA + SHA-256 · ECDSA + SHA-384. 그 밖(SHA-512 계열 · RSA-PSS 서명 인증서 · Ed25519 인증서 · P-521)을 만나면 **거절한다**. 폐기 확인도 없다. 모르는 알고리즘을 통과시키면 그때부터 이 모듈의 답은 «확인했다»가 아니라 «못 알아봤다»가 되기 때문이다.

체인은 마디의 줄이다

웹 서버가 보내는 인증서는 보통 세 층이다. 각 층은 바로 위 층에게 서명받고, 맨 위(뿌리)는 우리가 **미리 믿기로 한** 것이어야 한다.

```
뿌리 (신뢰 저장소 안)        ← trust.find_anchor 가 찾는다
   │ 서명
   ▼
중간 인증기관                ← link_ok 중간자 · 뿌리
   │ 서명
   ▼
잎 (example.com)             ← link_ok 잎 · 중간자  +  host_ok 잎 · "example.com"
```

`link_ok`를 아래에서 위로 마디마다 부르고, 잎에는 `host_ok`를 더한다. 하나라도 1 이 아니면 체인은 끊긴다. 체인을 **어떻게 모으는지**(서버가 보낸 순서, 몇 마디까지)는 부르는 쪽이 정한다 — 실제 예는 `apps/lowget`이다.

값	뜻
1	잇는다 — 서명 · CA · 이름 · 기간이 모두 맞다
2	자식의 발급자 이름이 부모의 주체 이름과 다르다
3	부모가 남을 발급할 수 있는 인증서(CA)가 아니다
4	둘 중 하나가 유효기간 밖이다
5	서명이 맞지 않는다(또는 모르는 알고리즘)
6	서명을 확인할 작업 공간이 모자랐다 — 잰 결과가 아니라 **재지 못한** 것이다

표 50.1 — `link_ok`가 내는 수 — 1 이 아니면 거절한 까닭

op

op	하는 일
`link_ok`	한 마디를 잇는다(위 표). 바이트 작업 공간 1024 · 조각 작업 공간 2200 이상

`signed_by`	자식이 부모에게 서명받았는가 — `ok 1` / `ok 0`, 자리가 모자라면 `error short_workspace`
`sig_kind`	자식이 적은 서명 알고리즘: 1 RSA+SHA-256 · 2 ECDSA+SHA-256 · 3 RSA+SHA-384 · 4 ECDSA+SHA-384 · 0 모른다
`dn_eq`	두 이름이 **바이트로** 같은가
`dates_ok`	유효기간이 지금(`YYYYMMDDhhmmss` 한 수)을 담는가
`host_ok` · `san_matches`	호스트 이름이 주체 대체 이름 중 하나와 맞는가
`ecdsa_rs`	DER 로 싼 ECDSA 서명에서 `r` · `s` 를 곡선 크기(32 · 48 바이트) 자리에 오른쪽 맞춤으로 꺼낸다
`curve` · `curve384`	P-256 · P-384 의 상수(`p` · `n` · `Gx` · `Gy`)를 작업 공간에 채운다

표 50.2 — `verify` 의 op

설계

서명은 원본 바이트 위에서 확인한다. `x509.tbs_off … tbs_end` 를 그대로 해시한다. 베낀 자리에서 확인하면 베끼기가 틀려도 서명이 맞아 보일 수 있다.

곡선은 공개키 길이가 가른다. 65 바이트면 P-256(`p256`), 97 바이트면 P-384(`p384`). RSA 는 `rsa` 로 간다.

이름은 바이트로 견준다. DER 은 한 이름에 한 바이트열이므로 그것이 옳다. 문자열로 풀어 견주면 대소문자 · 인코딩 · 공백 규칙이 끼어들고, 그 규칙마다 «같다» 의 뜻이 조금씩 달라진다.

와일드카드는 맨 앞 한 마디만. `*.a.b` 는 `x.a.b` 와 맞고 `a.b` 나 `y.x.a.b` 와는 맞지 않는다(RFC 6125 §6.4.3). 대소문자는 아스키로만 내린다.

`ecdsa_rs` 의 오른쪽 맞춤. DER 정수는 앞에 0 이 붙거나(최상위 비트가 1 이면) 짧을 수 있다. 그냥 베끼면 한 바이트 밀린 값으로 검증하게 되고, 그 실수는 언제나 «서명이 틀렸다» 로만 보인다.

모자람은 실패다. 작업 공간이 좁으면 `error short_workspace` 다 — 예전에는 «자리가 모자람» 과 «서명이 아님» 이 둘 다 0 이어서, 작업 공간을 잘못 준 프로그램이 «서명이 틀렸다» 로 보였다.

`tls13` — TLS 1.3 의 계산 부품

소스 `lib/tls13.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

TLS 1.3 이 요구하는 순수 계산 넷이다(RFC 8446) — **키 스케줄**(공유 비밀과 전사 해시에서 트래픽 키를 유도, §7.1), **레코드 계층**(레코드를 봉하고 연다, §5), **전사 해시**(지금까지 오간 메시지 전부의 해시, §4.4.1), **Finished**(핸드셰이크가 중간에 바뀌지 않았음을 증명, §4.4.4). 전부 순수해서 전송 없이 지을 수 있고 시험 벡터로 잴 수 있다 — 파서를 먼저, 전송을 나중에.

> **이것은 TLS 구현이 아니다**
>
> 상태 기계가 없다 — 핸드셰이크를 구동하지 않는다(메시지 층과 순서는 `tlssrv` 가 한다). 전송이 없다. PSK · 0-RTT · exporter · resumption 비밀이 없고 1-RTT 한 갈래만 지었다. PKI 가 없다(`der`). 상수 시간을 약속하지 않고 감사받지 않았다. 스위트는 둘뿐이고 모르는 스위트는 거절한다.

```
0 ─HKDF-Extract(PSK)→ Early Secret ─Derive-Secret("derived","")→ ┐
ECDHE ─HKDF-Extract──────────────→ Handshake Secret ←────────────┘
  ├─ Derive-Secret("c hs traffic", CH..SH)
  └─ Derive-Secret("s hs traffic", CH..SH)
─Derive-Secret("derived","")→ ┐
0 ─HKDF-Extract──────────────→ Master Secret
  ├─ Derive-Secret("c ap traffic", CH..server Finished)
  └─ Derive-Secret("s ap traffic", CH..server Finished)
```

각 비밀에서 레코드 키가 나온다 — `key = Expand-Label(비밀, "key", "", 길이)`, `iv = Expand-Label(비밀, "iv", "", 12)`.

op	하는 일
`build_label` · `expand_label`	HkdfLabel 구조체를 바이트로 · `HKDF-Expand-Label`
`derive_secret`	`Derive-Secret(비밀, 라벨, 전사 해시)`
`advance`	사다리 한 칸 — 위 그림의 화살표 하나
`traffic_key` · `traffic_iv`	비밀 → 레코드 키 · IV
`finished_key` · `verify_data`	Finished 의 키와 값
`record_header` · `record_nonce`	5 바이트 헤더 `23 ‖ 0x0303 ‖ 길이` · IV 와 시퀀스 번호 → 논스
`record_seal` · `record_open`	레코드 봉하기 · 열기
`inner_type`	속 평문의 **끝**에서 진짜 내용 타입을 읽는다
`transcript`	이어 붙인 메시지 버퍼의 해시
`hs_type` · `hs_size` · `hs_count`	핸드셰이크 메시지 걷기
`check_finished`	상대의 Finished 를 다시 계산해 맞춰 본다

표 50.1 — `tls13` 의 op

레코드는 겉과 속이 다르다. 겉은 언제나 `23 ‖ 0x0303 ‖ 길이` 다 — 내용이 핸드셰이크든 응용 데이터든 똑같이 보인다. 진짜 내용 타입은 속 평문의 끝에 있다. 관찰자에게 감추려고 그렇게 짰다. AAD 는 그 5 바이트 헤더 자체다.

스위트가 둘이라 협상이 뜻을 갖는다. 스위트 1 = `TLS_AES_128_GCM_SHA256`(MUST, `gcm`), 스위트 2 = `TLS_CHACHA20_POLY1305_SHA256`(SHOULD, `aead`). 하나만 지었을 때 이 자리는 죽은 분기였다. 모르는 스위트는 거절한다 — 조용히 하나를 고르지 않는다.

전사 해시가 핸드셰이크의 뼈대다. 키도 Finished 도 "지금까지 오간 메시지 전부" 의 해시 위에 서므로 중간자가 한 바이트라도 바꾸면 양쪽 키가 갈린다. **전사를 어디서 자르는지가 규격의 절반이다** — RFC 8448 이 각 Derive-Secret 의 해시로 그 자리를 적어 두었고, 시험이 세 자리(`CH…SH`, `CH…서버 Finished`, `CH…클라이언트 Finished`)를 전부 맞댄다. SHA-256 이 한 번에 하는 방식(스트리밍 없음)이라 전사는 **호출자가 이어 붙인 버퍼** 위에서 잰다. 메시지 걷기는 길이 필드를 믿지 않는다 — 버퍼를 넘는 길이가 적혀 있으면 거기서 멈춘다. `check_finished` 의 비교는 32 바이트를 XOR 로 누적해 마지막에 한 번 본다(조기 반환 없음).

작업 공간을 둘로 묶는 이유. 파라미터 상한이 16 이라 스위트마다 버퍼를 늘어놓으면 금방 넘는다. 그래서 바이트 `w` 와 `u64` `u` 둘로 묶고 자리를 소스 주석에 적어 나눠 쓴다. 예쁘지 않다 — 언어의 상한이 만든 모양이고, 그 사실을 감추지 않는 편이 낫다.

확인하는 것 — 두 겹이다. 파이썬 `hashlib` · `hmac` 로 §7.1 을 독립 구현해 바이트 대조하고, RFC 8448 §3 의 값들을 정본에서 기계로 뽑아 96 자리를 맞댄다. **그리고 정본이 결함을 하나 잡았다.** `traffic_key` 가 키 길이를 32 로 박아 두었는데, 그 길이는 HkdfLabel 안에 들어가므로 AES-128-GCM(16)을 쓰는 상대와는 키가 통째로 달라진다. 두 번째 구현으로는 잡히지 않았다 — 같은 사람이 같이 32 를 썼기 때문이다. 두 번째 구현은 정본이 아니다.

tlssrv — TLS 1.3 서버 핸드셰이크

소스 lib/tlssrv.low
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

tls13 이 규격의 **계산**을 갖고 있다. 이 모듈은 그것을 **순서대로 부르는 층**이다 — ClientHello 를 읽고, 서버 메시지를 짓고, 상태를 옮기고, 응용 데이터 레코드를 봉하고 연다. 모든 실패는 0 이고, op 은 **자리**를 낸다 — 자르는 것은 부르는 쪽이 subslice 로 한다.

이것만으로는 TLS 서버가 아니다

> **전송이 없다** — 소켓도 재조립도 없고 바이트열은 부르는 쪽이 모아 온다(레코드 경계도 부르는 쪽이 정한다). 키 갱신(KeyUpdate) · 레코드 패딩 · HelloRetryRequest · PSK · 0-RTT · 클라이언트 인증서 요구 · 세션 재개가 없다. 인증서 체인은 build_cert 가 하나만 담는다. 확장은 **찾아만** 준다(SNI · ALPN 해석은 부르는 쪽). 감사받지 않았다.

ClientHello 는 가장 적대적인 입력이다. 아직 아무것도 인증되지 않은 바이트열이고 길이 필드는 전부 상대가 썼다. 그래서 — 어떤 자리도 버퍼를 넘으면 즉시 실패한다. 확장 걷기는 재귀하지 않고 걸음수를 버퍼 크기로 묶는다(길이 0 짜리 확장이 이어지면 묶지 않은 루프는 끝나지 않는다). 길이가 맞지 않으면 고쳐 읽지 않는다 — ch_ok 는 선언된 길이가 버퍼와 **정확히** 같기를, ch_ext_len 은 확장이 끝까지 맞기를 요구한다.

op	하는 일
st_start · st_recvd_ch · st_negotiated · st_wait_flight2 · st_wait_finished · st_connected	상태 번호
hs_client_hello · hs_server_hello · hs_encrypted_extensions · hs_certificate · hs_certificate_verify · hs_finished	메시지 종류 번호
next_ok · step	지금 이 메시지를 받아도 되는가 · 상태를 옮긴다(못 옮기면 같은 상태를 낸다)
ch_ok	종류와 길이가 버퍼와 맞는가
ch_random_off · ch_sid_off · ch_sid_len	랜덤 · session_id
ch_suites_off · ch_suites_len · ch_has_suite	제안된 스위트
ch_ext_off · ch_ext_len · ch_ext_find · ch_ext_find_len	확장 블록 · 종류로 찾기
ch_x25519_off · be16	key_share 안의 x25519 공개키 자리 · 빅엔디언 2바이트
build_sh · build_ee · build_cert	ServerHello · EncryptedExtensions · Certificate(DER 하나) 짓기
cv_content · build_cv · build_fin	CertificateVerify 가 서명하는 130 바이트 · CertificateVerify · Finished 짓기
server_finished	ECDHE 부터 키 스케줄을 올려 서버 Finished 까지 한 줄로
app_secrets · check_client_finished	응용 트래픽 비밀(c · s) · 상대 Finished 확인(1 = 맞음)

`traffic_keys` · `seal_app` · `open_app`	비밀 → 키 ‖ IV(한 방향씩) · 응용 데이터 레코드 봉하기 · 열기

표 50.1 — `tlssrv` 의 op

메시지 짓기는 틀이 같다(`<종류 1> <길이 3> <본문>`) — 그래서 틀을 한 번만 적었다. 네 곳에 같은 산술을 되풀이하면 한 곳만 고치는 날이 온다.

`cv_content` — **공백 64 개는 장식이 아니다.** 서명 대상은 `0x20 × 64 ‖ "TLS 1.3, server CertificateVerify" ‖ 0x00 ‖ 전사 해시` 다(§ 4.4.3). 그 앞머리가 없으면 이 서명이 다른 문맥(인증서 서명, 클라이언트 쪽 서명)의 서명으로 재활용될 수 있다. 규격에서 "왜 이런 게 있지" 싶은 상수는 대개 이미 일어난 공격의 흔적이다.

순서는 규격의 절반이다. `next_ok` 는 받아들이는 자리를 열거하고, 열거하는 것이 곧 나머지를 거절하는 것이다. 보지 않으면 중간자가 Finished 를 앞당기거나 ClientHello 를 두 번 보낼 수 있다. **`session_id` 는 그대로 되울린다** — TLS 1.3 은 그 필드를 쓰지 않지만, 되울리지 않으면 1.2 로 보이게 하려는 호환 장치 때문에 실제 망의 중간 상자에서 끊긴다(§ 4.1.3).

전사는 이 층에서 세 번 잘린다. `s hs traffic` 은 `CH‖SH`, 서버 Finished 는 `CH‖…‖CertificateVerify`, 응용 트래픽 비밀과 **클라이언트 Finished** 는 `CH‖…‖서버 Finished` 위에 선다. 클라이언트 Finished 는 자기 자신을 포함하지 않고, 그 verify_data 는 **클라이언트** 핸드셰이크 비밀에서 나온다 — 같은 전사, 다른 비밀. 서버 것을 쓰면 언제나 거절하게 되고 증상은 "클라이언트가 이상하다" 로 보인다.

응용 데이터는 방향마다 다른 비밀 · 다른 시퀀스다. 하나를 공유하면 논스가 겹치고, AEAD 에서 논스가 겹치는 것은 평문과 인증키를 함께 잃는 일이다. 그래서 키뭉치를 한 방향씩 만든다. 시퀀스는 **레코드마다** 오르고 키 세대마다 0 부터다. `traffic_keys` 의 출력 버퍼는 키가 16 이어도 32 바이트여야 한다 — `expand_label` 은 HMAC 한 블록을 쓰므로 언제나 32 를 쓴다. 경보도 같은 레코드다 — 속 타입이 21 이고, 그것을 보지 않으면 경보를 데이터로 읽는다.

확인하는 것 — RFC 8448 §3 의 실제 핸드셰이크. 정본의 ClientHello 필드를 읽고, 메시지 다섯을 지어 정본과 바이트로 맞댄다(ServerHello 90 · EncryptedExtensions 40 · Certificate 445 · CertificateVerify 136 · Finished 36). 그리고 끝까지 엮는다 — 정본의 키로 ECDHE 를 우리가 내고 키 스케줄을 올려 서버 Finished 를 정본과 맞춘다. 이 한 수가 전사 자르는 자리 · 키 스케줄 · finished_key · verify_data 를 한꺼번에 잰다. 정본의 CertificateVerify 는 RSA-PSS 이고 우리 서명기는 ECDSA 라, 여기서 재는 것은 틀이고 서명 자체는 `ecdsa` 가 따로 잰다.

`tlscli` — TLS 1.3 클라이언트 핸드셰이크

소스 `lib/tlscli.low`
층 L0 — 순수 계산(호출자의 뒷받침)
권한 없음

`tlssrv` 의 **거울상**이다. 서버 쪽은 ClientHello 를 읽고 ServerHello 를 짓는다. 이 모듈은 반대로 ClientHello 를 **짓고** ServerHello 이하를 **읽는다**. 규격의 계산(키 스케줄 · 레코드 봉인)은 `tls13` 에 있고, 이 모듈은 그것을 클라이언트의 순서대로 부르는 층이다.

> **이것만으로는 TLS 클라이언트가 아니다**
>
> **전송이 없다** — 소켓도 재조립도 없고, 바이트열은 부르는 쪽이 모아 온다. 그래서 이 모듈은 `effects none` 이다. **인증서를 확인하지 않는다** — `Certificate` 메시지에서 인증서를 꺼내 줄 뿐이고, 체인 · 유효기간 · 이름은 `x509` · `verify` · `trust` 로 부르는 쪽이 확인한다. 확인하지 않는 도구는 그렇다고 말해야 한다(`lowget` 의 `--insecure`). 짓지 않은 것: HelloRetryRequest · PSK/0-RTT · 세션 재개 · 클라이언트 인증서 · x25519 가 아닌 키 교환 그룹 · `TLS_CHACHA20_POLY1305_SHA256` · `TLS_AES_128_GCM_SHA256` 이 아닌 스위트.

핸드셰이크의 차례

클라이언트는 한 가지 차례로만 메시지를 받는다. 이 차례를 지키지 않으면 중간자가 메시지를 빼거나 바꿔치기할 수 있다. `cnext_ok` 가 상태마다 **받아도 되는 메시지 하나**를 열거하고, 그 밖은 모두 거절한다.

```
cst_start ──ClientHello 보냄──▶ cst_wait_sh
cst_wait_sh ──ServerHello(2)──▶ cst_wait_ee        — 여기서 핸드셰이크 키를 만든다(hs_secrets)
cst_wait_ee ──EncryptedExtensions(8)──▶ cst_wait_cert
cst_wait_cert ──Certificate(11)──▶ cst_wait_cv     — cert_at 으로 꺼내 부르는 쪽이 확인
cst_wait_cv ──CertificateVerify(15)──▶ cst_wait_finished
cst_wait_finished ──Finished(20)──▶ cst_connected  — check_server_fin 이 참이어야 한다
```

괄호 안의 수는 핸드셰이크 메시지 종류 번호다. `cstep` 이 다음 상태를 낸다.

op

op	하는 일
`cst_start` … `cst_connected`	상태 번호(0 … 6)
`cnext_ok` · `cstep`	지금 상태에서 이 메시지를 받아도 되는가 · 그다음 상태
`build_ch`	ClientHello 를 짓는다 — 스위트는 ChaCha20-Poly1305 를 먼저, AES-128-GCM 을 다음에, 키 교환은 x25519
`sh_ok` · `sh_usable`	ServerHello 가 성한가 · 우리가 이어 갈 수 있는 것인가(한 번에 판정)
`sh_is_hrr`	HelloRetryRequest 인가 — 알아보고 **거절한다**
`sh_suite` · `sh_is_tls13` · `sh_key_share_off`	고른 스위트 · 정말 1.3 인가 · 서버의 x25519 공개값 자리

`sh_ext_off` · `sh_ext_len` · `sh_ext_find` · `sh_ext_find_len`	확장 묶음과 확장 하나 찾기
`hs_secrets`	핸드셰이크 비밀을 만든다(키 사다리)
`finished_vd`	어떤 트래픽 비밀로든 Finished 의 검증값을 낸다
`check_server_fin`	서버의 Finished 를 검산한다 — «상대가 그 비밀을 정말 갖고 있는가»
`build_client_fin`	우리 Finished 를 짓는다
`seal_rec`	레코드 하나를 봉한다(속 타입을 부르는 쪽이 정한다)
`plain_hdr` · `rec_len` · `rec_type`	평문 레코드 머리를 짓는다 · 받은 레코드 머리를 읽는다
`cert_at` · `cert_len`	Certificate 메시지에서 `n` 번째 인증서(0 이 잎)의 자리와 길이

표 50.1 — `tlscli` 의 op

설계

읽는 쪽 길이는 전부 상대가 쓴 것이다. ServerHello 이하는 아직 아무것도 인증되지 않은 바이트열이다. 어떤 자리도 버퍼를 넘으면 곧바로 0 을 답하고, 길이가 안 맞으면 고쳐 읽지 않는다.

HelloRetryRequest 를 ServerHello 로 읽지 않는다. HRR 은 ServerHello 와 같은 메시지 종류이고 랜덤 자리에 정해진 32 바이트가 들어가는 것으로만 구분된다. 그것을 모르면 HRR 의 랜덤을 진짜 랜덤으로 읽어 키 사다리를 **조용히 틀린 값**에서 쌓는다. HRR 을 짓지 않기로 했으므로 알아보고 거절한다.

판을 고르는 것은 머리가 아니라 확장이다. 메시지 머리의 `0303` 은 중간 상자를 속이려는 장식이다. 정말 1.3 인지는 `supported_versions` 확장이 `0304` 인지로 본다(`sh_is_tls13`).

왜 ChaCha20 을 먼저 제안하나. 이 언어로 쓴 두 암호의 처리량을 재어 빠른 쪽을 앞에 둔다. 서버가 순서를 존중하면 빠른 쪽을 고른다.

`check_server_fin` 이 참이어도 «누구인지» 는 아직 모른다. 그것은 «상대가 핸드셰이크 비밀을 갖고 있는가» 를 확인할 뿐이고, 그 상대가 누구인지는 인증서 쪽(`verify`)의 물음이다. 둘을 모두 확인해야 연결을 믿을 수 있다.

왜 이 모듈이 있나. 클라이언트가 있어야 바깥의 진짜 구현(`openssl s_server`, 실제 웹 서버)이 반대편에 설 수 있다. 우리 서버(`tlssrv`)와만 맞대면 양쪽이 같은 오해를 나눠 가질 수 있고, 그러면 «규격을 같게 읽었다» 는 확인이 되지 않는다. 이 모듈 위에 선 실물 도구가 `apps/lowget` 이다.

`http` — HTTP/1.1 요청 파서

소스 `lib/http.low`
층 L0 — 순수 계산
권한 없음

요청 바이트열을 받아 각 필드가 **어디서 시작해 몇 바이트인지** 답한다(RFC 9112). 자르지 않는다 — 자르는 것은 부르는 쪽이 `subslice` 로 한다. **모든 실패는 `0`** 이다. `0` 은 정당한 자리가 될 수 없다 — 요청은 최소 `GET / HTTP/1.1␍␊␍␊` 라 어떤 필드도 0 에서 시작하지 않는다.

무엇을 약속하고 무엇을 하지 않나

서버 쪽 요청 파싱만 한다(응답 파싱 없음). 전송이 없다 — 바이트열은 부르는 쪽이 모아 온다. **청크 전송(`Transfer-Encoding: chunked`)을 짓지 않았다** — 그 헤더를 찾아 주기만 하므로 부르는 쪽이 거절해야 한다. 트레일러 · 여러 줄 헤더(obs-fold) · URL 퍼센트 해체 · HTTP/2 가 없다. HTTP/1.0 을 받지 않는다 — 지속 연결 규칙이 다르고 그것을 짓지 않았다.

```
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	하는 일
`method_get` · `method_head` · `method_post` · `method_put` · `method_delete`	메서드 코드(1 … 5)
`method_code`	요청 줄의 메서드 → 코드. 0 = 모름
`line_next` · `line_len`	다음 줄의 자리 · 이 줄의 내용 길이(CRLF 제외)
`target_off` · `target_len`	요청 대상의 자리 · 길이
`version_ok`	`HTTP/1.1` 인가
`headers_off` · `header_next`	첫 헤더 · 다음 헤더(빈 줄이면 0)
`name_len` · `value_off` · `value_len`	헤더 이름 길이 · 값 자리 · 값 길이
`name_eq`	이 헤더의 이름이 그것인가(대소문자 안 가림)
`header_find` · `header_find_len`	그 이름의 값 자리 · 길이. **중복이면 0**
`content_length`	`option u64` — 없으면 `some 0`, 성하지 않으면 `none`
`body_off`	몸통이 시작하는 자리

표 50.1 — `http` 의 op

이 모듈의 알맹이는 거절이다. 파서는 받아들이는 것보다 거절하는 것으로 정의된다. HTTP 에서 잘못 받아들이는 자리에는 이름이 있다 — **요청 밀반입(request smuggling)**. 앞단(프록시)과 뒷단(서버)이 같은 바이트를 다르게 읽으면 하나가 본 요청을 다른 하나는 못 본다.

거절하는 것	왜
맨 `LF` 를 줄 끝으로	앞단이 CRLF 만 인정하면 경계가 갈린다
이름과 콜론 사이 공백(`Host : x`)	RFC 9112 §5.1 이 거절을 요구한다
`Content-Length` 가 둘	값이 같아도 거절한다
`Content-Length: 5, 5` · `+5` · 빈 값	숫자만 받는다 — 그 관용이 밀반입이다

HTTP/1.0	지속 연결 규칙이 다른데 짓지 않았다
같은 헤더가 둘(header_find)	합쳐도 되는지는 헤더마다 다르고 그 표를 짓지 않았다 — 모르면 거절
빈 대상(GET HTTP/1.1) · 끝나지 않은 헤더(빈 줄 없음)	—

표 50.2 — http 가 거절하는 것

content_length 가 option 인 이유. “없다” 와 “틀렸다” 는 다른 답이다. 없으면 some 0(몸통이 없는 정상 요청), 성하지 않으면 none(연결을 끊어야 할 일). 한 값으로 두면 둘이 섞이고, 섞이는 자리가 곧 공격 자리다.

이 모듈이 처리기 결함을 하나 잡았다. 처음에 input b str . 로 썼다. --check 는 통과했는데 21 op 중 16 이 느린 해석 경로로 떨어졌다(약 80 배). str 은 빌트인이 아니라 strings 의 지역 별칭이었고, 검사기는 그 이름을 통과시켰지만 타입 붙은 하강은 뜻을 몰랐다. 답은 맞으므로 시험이 영원히 보지 못하는 조용한 80 배다. 이제 그런 이름을 시그니처에 쓰면 W-NOT-YET 경고가 나온다. 검사가 통과하는 것과 빠른 것은 다른 일이고, --why-slow 를 보지 않았으면 이 파일은 그대로 실렸을 것이다.

확인하는 것 — RFC 9112 예제(정상 8), 거절 9 건, VM·네이티브 일치, 느린 경로로 떨어지는 op 0 개.

soa — SoA 배치 시범: 필드마다 배열 하나

소스 `lib/soa.low`
층 L0 — 순수 계산
권한 없음

원소가 `{x, y, vx, vy}` 인 자료 n 개를 두는 방법은 둘이다. **AoS**(Array of Structs)는 한 배열에 원소를 통째로 늘어놓고, **SoA**(Struct of Arrays)는 **필드마다 배열 하나**를 나란히 둔다. x 만 훑는 계산이라면 SoA 쪽이 필요한 값만 연속으로 읽어 캐시를 알뜰하게 쓴다.

답이 아니라 측정이다

SoA 를 언어 기능(`store[T, soa]` 같은 타입 생성자와 키워드 · op 수십 개)으로 넣자는 조사는 **보류**됐고, 재검토 조건은 "먼저 `lib/` 에 빌트인 증가 0 으로 써 본다. 거기서 막히는 곳이 언어 작업의 목록이 된다" 였다. 이 모듈이 그것이다. 발견 — **SoA 배치 자체는 라이브러리로 완전히 표현된다**. 필드별 배열, 보폭 없는 순차 접근, 한 필드만 훑는 커널이 전부 새 언어 기계 없이 써진다. 속도 주장은 측정 없이 하지 않는다 — 이 모듈이 보장하는 것은 두 배치가 같은 답을 낸다는 정확성뿐이고, 원소 전체를 만지는 코드는 AoS 가 낫다.

op	모양	실패
`step_x`	`proc (xs mut slice u64, vxs slice u64, n u64) → u64` — `xs[i] += vxs[i]`	없음 — `min(n, len xs, len vxs)` 만 처리하고 그 수를 답한다
`sum_field`	`fn (f slice u64) → u64`	없음
`get_x`	`fn (xs slice u64, i u64) → u64`	범위 밖이면 `0`
`step_all`	`proc (xs, ys mut slice u64, vxs, vys slice u64, n u64) → u64`	없음 — 두 처리 수 중 작은 쪽
`step_x_aos`	`proc (rows mut slice u64, stride, xoff, voff, n u64) → u64` — AoS 판	속도 자리가 범위 밖인 원소에서 멈추고 그 `i` 를 답한다

표 50.1 — `soa` 의 op — 모두 `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
```

막히는 자리 — 언어 작업의 목록. ① 원소 하나를 "한 덩어리" 로 다루는 문법이 없다 — 호출자가 필드를 손으로 모은다(`get_x`). 불편할 뿐 불가능하지 않다. ② 필드 개수만큼 인자가 늘어난다(`step_all`) — 이것은 구조체 필드에 슬라이스를 허용하면서 풀렸지만, 이 모듈은 측정 기록이라 네 인자 모양을 그대로 둔다. ③ **타입이 배치를 모른다** — AoS 판과 SoA 판이 서로 다른 op 이름이 되고(`step_x` 대 `step_x_aos`), AoS 의 오프셋 인자는 전부 `u64` 라 컴파일러가 지켜 주지 못한다.

> **반례. 반환된 처리 수를 보지 않는다**
>
> `soa.step_x xs vxs 1000` 은 `xs` 가 3 칸이면 조용히 3 개만 처리한다. `n` 개가 전부 처리됐다고 가정하는 코드는 `guard eq m n .` 으로 확인한다.

> **반례. AoS 판의 오프셋을 바꿔 낀다**
>
> `soa.step_x_aos rows 2 1 0 3` 은 xoff 와 voff 가 뒤집혀 에러 없이 속도에 위치가 더해진다. 배치가 타입에 실리지 않는다는 막힘의 실감이다.

주의. 나란한 배열들의 길이를 맞추는 것은 호출자 책임이다 — op 은 짧은 쪽에 맞춰 줄일 뿐 알려 주지 않는다. `get_x` 의 실패 값 0 은 정상 값과 구분되지 않는다. 맨 `index` 는 범위를 줄여 주지 않고 `E-VM-BOUNDS` 로 멈춘다. 긴 `let · set` 을 줄바꿈으로 나누면 개행이 form 을 닫는다 — 이어 쓰려면 줄 끝에 `,` 를 둔다.

allocs — 얼로케이터 trait 과 범프 · 기본 얼로케이터

소스 lib/alloc.low
층 L1 — 저장
권한 범프는 없음 · fixed_bytes 는 cap allocator · heap_bytes 는 cap heap 을 쥔 곳에서만 띄운다

얼로케이터의 **인터페이스**(trait 둘)와 그것을 충족하는 actor 넷이다. 가장 흔한 쓰임은 빌린 큰 바이트 덩어리를 **앞에서부터 잘라 주는** 범프다 — "만들 때는 여럿, 버릴 때는 한꺼번에" 인 자료(파서의 임시 노드, 한 처리 동안만 사는 문자열)에 맞는다(35장).

```
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 .
```

뿌리와 그 위. Lowent 에는 암묵 전역 힙이 없다. 바이트가 프로그램에 처음 들어오는 자리는 뿌리 op alloc_bytes <권한> capacity n 하나뿐이고, 뿌리는 둘이다 — **고정 창**(cap allocator · 효과 alloc, 베어메탈에서는 링커가 창의 경계를 정한다)과 **힙**(cap heap · 효과 heap, 운영체제가 있는 호스티드에만 있다)(20장). allocs 는 그 뿌리 **위**를 맡는다. 받은 바이트가 뿌리에서 왔든 호출자가 빌려준 것이든 묻지 않는다. 그래서 cap allocator 를 받지 않은 코드도 남이 준 바이트 위에서는 온전히 할당한다 — 권한이 능력을 가른다. actor · trait · option · subslice 만으로 쓰이므로 빌트인이 아니라 라이브러리다.

```
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 는 같은 셋에 release input s self . input v mut slice u8 . . output bool . 을 더한다. via self 는 "이 op 의 효과는 구현이 적은 효과다" 라는 뜻이다 — 범프의 reserve 는 state 뿐이고 heap_bytes 의 reserve 는 heap state 다. 그래서 그 얼로케이터로 단형화한 컨테이너 op 의 서명에 heap 이 선다.

actor	받침 · 정책	reserve 의 효과 · 쓸 수 있는 곳
bump_bytes	init 으로 건 버퍼 · 커서를 요청 크기만큼 민다 · 두 trait 모두	state · 어디서나
bump_aligned	같은 받침 · 시작점을 8 의 배수로 올린 뒤 자른다(패딩은 버려진다)	state · 어디서나
fixed_bytes	권한 칸 root cap allocator · 뿌리에서 곧장 깎는다	alloc state · 어디서나(베어메탈은 링커 창)
heap_bytes	권한 칸 root cap heap	heap state · 호스티드만(베어메탈은 E-HEAP-NOHOST)

표 50.1 — allocs 의 actor

op	모양	실패
init(trait 밖)	backing mut slice u8 → u64(용량), effects state	없음. 다시 부르면 새 버퍼로 갈아타고 커서가 0
reserve	n u64 → option mut slice u8	none — 커서는 움직이지 않는다(부분 할당 없음)

grow	old mut slice u8, newn u64 → option mut slice u8	none — 마지막 조각이 아니거나 자리 부족
release	v mut slice u8 → bool	false — 마지막 조각이 아니면 아무것도 바꾸지 않는다
used	→ u64(커서 위치, 패딩 포함), effects none	없음

표 50.2 — 범프의 op

- **OOM 은 값이다.** reserve 는 멈추는 대신 none 을 돌려준다. guard is_some … 없이는 바이트에 닿을 수 없다.
- **돌려주는 것은 오프셋이 아니라 뷰다.** reserve 의 결과에 쓰면 뒷받침 버퍼가 바뀐다.
- **grow 는 마지막 조각을 제자리에서 늘린다.** 판정은 "건넨 조각이 마지막으로 준 **바로 그 바이트**인가"(same_slice)다. 전에는 길이만 보았고, 길이만 같은 남의 버퍼가 통과해 두 컨테이너가 조용히 겹쳤다(보안 검토에서 드러났다). **실패는 값이고, 이것은 최적화이지 계약(op 이 스스로 적는 약속)이 아니다** — 새 얼로케이터는 grow 를 return none . 한 줄로 둬도 완전하다. 이것으로 성장 벡터의 아레나 고수위가 요청의 약 4 배에서 2 배가 됐다(growvec).
- **release 는 마지막 조각만 받는다** — 모르는 조각을 조용히 받아 두면 두 번 돌려주기가 남의 자리를 지우기 때문이다.
- fixed_bytes · heap_bytes 는 init 이 없고 grow 는 언제나 none 이다(뿌리는 마지막 할당이 누구 것인지 모른다). 권한 칸은 실행 중 값이 아니다 — 그 actor 를 spawn 하는 op 이 같은 종류의 권한을 쥐고 있어야 한다(E-CAP-FORGE). 권한 없는 곳에서 한 줄로 힙을 지어낼 수 없다.

얼로케이터를 받는 코드는 input comptime a type . + using al a . + requires allocs.byte_allocator a . 로 어느 구현이든 받는다. 부르는 쪽은 위치 인자가 아니라 let x … using <출처> be … 로 건네고, 그 op 안에 출처가 하나뿐이면 적지 않아도 기본값이 된다. 단형화되므로 vtable 도 간접 호출도 없다.

```
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
```

범프를 주면 3 + 5 를 9 로 늘려 12, bump_aligned 를 주면 둘째 조각이 8 에서 시작해 17, heap_bytes 를 주면 grow 가 none 이고 인스턴스 서명에 heap 이 선다.

> **반례. none 검사 없이 값을 꺼낸다**
>
> some_value (send a reserve 99) 는 버퍼가 작으면 E-VM-NONE 으로 멈춘다. 컴파일은 통과하므로 방심하기 쉽다.

반례. 권한 없이 뿌리에 닿는다 · fn 에서 send 한다

> 뿌리 권한을 받지 않은 op 은 `alloc_bytes` 를 부를 수 없다(`E-ALLOC-NOCAP`, 힙이면 `E-HEAP-NOCAP`) — `allocs` 는 그 대체가 아니다. `effects none` 인 `fn` 에서 핸들러를 부르면 `E-EFFECT-CALC` 다. trait 을 충족하지 않는 값을 출처로 건네면 `E-BOUND-UNSAT` 이다.

주의. 마지막 조각 말고는 해제가 없다 — 받고 놓기를 되풀이하는 모양이면 `pool` 이 맞다. 스코프 기반 일괄 해제는 `region` 블록의 몫이다 (18장). `reserve` 가 낸 뷰는 뒷받침 버퍼의 별칭이라 `init` 을 다시 불러도 사라지지 않는다. actor 는 순차 배달 전제다 — 범프를 태스크에 건네면 거절된다 (`E-ALLOC-SHARED`: 태스크에 건네는 얼로케이터는 `reserve` 가 `atomic` 이어야 한다). 바닥이 0 에서 시작하고 주소를 노출하지 않으므로 VM 과 네이티브가 같은 바이트를 본다.

pool — 세대 핸들 블록 풀

소스 lib/pool.low
층 L1 — 호출자의 저장
권한 없음

고정 크기 블록을 빌려 주고 돌려받는 풀이다. 핸들에 **세대 번호**가 붙어 옛 핸들을 알아본다. 객체를 만들고 지우기를 되풀이할 때, 이미 지운 것을 실수로 다시 쓰는 결함(use-after-free)을 막고 싶을 때 쓴다. "필요할 때 4 KiB 씩 받아 쓰고, 원하는 시점에 특정 블록을 해제한다" 는 해제 시점이 어휘적이지 않으므로 region 으로도 범프로도 안 된다 — 그래서 정적 검사 대신 세대 핸들로 간다(18장, 35장).

```
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 .
```

무엇을 막아 주고, 무엇을 안 막아 주나

> **막아 준다 ①** — 돌려준 블록에 **옛 핸들로 접근**하는 것. 세대가 달라 거절된다(실행 중 값으로). **막아 준다 ②** — **풀을 섞는 것**. 핸들과 풀이 **브랜드**를 타입으로 들기 때문에 handle pa 를 block_pool pb 에 넣으면 컴파일 에러 E-TYPE-INSTANCE 다. 그리고 mem · gens 는 init 이 봉해 들어 op 이 더 이상 받지 않는다 — 엉뚱한 배열을 건네는 길이 표면에서 사라졌다. 브랜드는 값이 아니라 타입이라 핸들 칸은 늘지 않는다. **막아 주지 않는다** — 브랜드는 선언마다 하나다. 브랜드를 comptime 인자로 받아 init 하는 op 을 두 번 부르면 한 브랜드가 풀 둘을 덮는다. 풀 하나에 브랜드 하나를 지키는 것은 당신의 몫이다. 이것은 언어에 내장된 세대 핸들이 아니다 — 평범한 라이브러리이고, 안전은 이 모듈의 규율에서 나온다.

원리. 블록마다 세대 번호를 두고, 받을 때 핸들에도 그 번호를 적어 준다. 해제하면 블록의 세대가 올라가고, 그 순간 옛 핸들은 번호가 맞지 않아 자동으로 무효가 된다. **고정 크기**라 해제가 자유 목록에 넣는 것이 전부이고 단편화가 0 이다. 세대는 나란한 배열 gens 에 살고(SoA), 검사 비용은 bytes 라는 문을 지나는 코드에만 붙는다. **자유 목록 링크는 블록 자기 바이트 안에 산다** — 해제된 블록은 세대가 올라 아무도 닿지 못하므로 그 앞 8 바이트를 장부로 쓰는 것이 공짜다. 풀은 actor 가 아니라 struct 다(actor state 는 슬라이스를 들 수 없다).

op	모양	실패
handle b • block_pool b	구조체 — 핸들은 blk · len · gen, 풀은 봉인된 mem · gens 와 커서들	—
init	comptime b, mem mut slice u8, g mut slice u64, bs u64 → option (block_pool b)	bs < 8 이면 none
blocks	fn (comptime b, p) → u64 — min(len mem / bs, len g)	없음
take	(comptime b, p mut block_pool b) → option (handle b)	블록이 없으면 none
release	(comptime b, p mut, h handle b) → bool	낡은 · 범위 밖 핸들(이중 해제 포함)이면 false
bytes	(comptime b, p, h) → option mut slice u8 — 바이트에 닿는 유일한 문	낡은 · 범위 밖이면 none
alive	(comptime b, p, h) → bool	없음(거짓이 답)

`used • outstanding`	→ `u64` — 순차로 꺼내 본 블록 수 · 지금 밖에 나가 있는 블록 수	없음

표 50.1 — `pool` 의 op — 첫 인자가 브랜드, 모두 `effects none`

`take` 는 **놓은 것을 먼저 준다**(자유 목록은 LIFO) — 받고 놓기를 되풀이해도 풀이 마르지 않는다. 재사용된 블록의 핸들은 세대가 올라간 새 핸들이다. 브랜드를 매번 적는 이유 — 이 언어에는 추론되는 타입 파라미터가 없고, 그 한 낱말이 "이 핸들은 저 풀의 것" 이라는 계약이다.

뷰는 빌림 머리에서 꺼낸다 — `bytes` 가 준 슬라이스는 평범한 슬라이스라 반환을 모른다. 그래서 뷰는 `borrow v be some_value (pool.bytes <브랜드> p h) do … end` 처럼 **빌림 머리에서** 꺼내고, 바이트는 빌림 안에서만 만지고, 해제는 블록 **뒤에** 한다(12장). 이렇게 꺼내면 처리기가 빌림 동안 그 풀을 쓰기 자리에 넘기는 것(`release`·`take`)과 같은 풀로 두 번째 빌림을 여는 것을 `E-BORROW-EXCL` 로 거절한다.

```
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
```

반례. 해제한 핸들로 다시 닿는다 · 이중 해제

해제 뒤 `bytes` 는 `none` 이고, 검사 없이 `some_value` 를 부르면 그 줄에서 `E-VM-NONE` 으로 멈춘다. 두 번째 `release` 는 조용히 `false` 를 돌려줄 뿐이라, 반환값을 보지 않으면 "놓았다고 믿었는데 놓이지 않은" 결함이 숨는다. `guard eq rel true .` 로 받는다.

반례. 블록 앞 8 바이트에 남아야 할 값을 두고 해제한다

해제가 그 자리를 자유 목록 링크로 덮는다. 세대가 올라 아무도 닿지 못하므로 안전하지만, "해제 뒤에도 메모리에 남아 있겠지" 라는 기대는 앞 8 바이트에서 틀린다. 자료는 8 번 바이트부터 둔다.

반례. 빌린 이름을 블록 밖으로 내보낸다

`borrow v be bv do set out v . end` 는 컴파일 에러 `E-BORROW-ESCAPE` 다 — 빌림은 블록 끝에서 끝난다.

반례. 빌림 안에서 해제한다

`borrow v be some_value (pool.bytes b p h) do pool.release b p h . … end` 는 컴파일 에러 `E-BORROW-EXCL` 이다 — 반환은 풀을 고치고, 빌린 뷰는 그것을 모른다. 해제는 블록을 닫은 뒤에 한다.

반례. 해제한 뒤 옛 뷰를 쓴다

`pool.release` 는 `invalidates p .` 를 밝힌다. 그래서 `let bv … be some_value (pool.bytes b p h) .` 로 묶어 둔 뷰를 해제 **뒤에** 쓰면 컴파일 에러 `E-VIEW-INVALIDATED` 다 — 처리기가 `bv` 가 `p` 에서 왔다는 것을 따라간다. 해제 뒤에 다시 닿으려면 `bytes` 로 새로 묻는다 (세대가 `none` 으로 답한다).

반례. 같은 핸들로 쓰기 뷰 둘을 빌림 밖에서 든다

`bytes` 를 같은 인자로 두 번 불러 이름 둘에 묶으면 한 블록에 쓰는 이가 둘이다. 빌림 머리에서 꺼내면 두 번째 빌림이 `E-BORROW-EXCL`, 빌림 밖에서 묶은 둘을 함께 쓰면 `E-EXCL` 로 거절된다. 뷰는 하나만 꺼내 빌림 머리에서 쓴다.

주의. 핸들은 값이라 복사해 들고 다닐 수 있지만, 어느 복사본으로든 `release` 하면 전부 낡는다. 받고 놓기를 아무리 되풀이해도 `outstanding` 이 늘지 않는 것이 자유 목록이 사는 증거다(`used` 는 처음 몇 라운드만 는다). `outstanding` 은 자유 목록을 걸으므로 O(자유 블록 수)다 — 뜨거운 경로에서 매번 부르지 않는다. 한 핸들 = 한 블록이다. 순차 배달 전제다. `op` 이름이 `live` 가 아니라 `outstanding` 인 이유 — 흔한 낱말은 처리기가 지역 변수와 `op` 머리를 가르지 못했다. 핸들의 비트 폭 설계는 `budget` 이 돕는다.

shard — 저장소를 쪼개는 접근 단위

소스 lib/shard.low
층 L1 — 호출자의 저장
권한 없음

저장소 한 덩어리를 **서로 겹치지 않는 조각**으로 나눠 각 조각을 따로 바꿀 수 있게 한다. 조각을 대표하는 것이 **토큰**이고, 토큰은 "이 범위는 내 것" 이라는 권한이다. 그것을 여러 스레드에 태우든 순차로 돌든, 겹치지 않는다는 사실 자체가 값이다.

```
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 이 뒤로 r 은 쓸 수 없다 --- 쓰면 E-OWN-MOVED
```

막아 주는 것 — 그리고 누가 막는가

① 쪼갠 뒤 root 로 만지는 것은 컴파일 에러 E-OWN-MOVED 다. 그것을 막는 것은 이 모듈이 아니라 언어다 — 토큰이 owned 라서 split_at 에 넘기는 순간 손을 떠난다. ② 남의 조각 자리를 만지면 write · read 가 false · none 으로 답한다(멈추지 않는다 — 실패는 값이다). ③ 다른 저장소의 토큰은 브랜드가 타입으로 달라 섞이지 않는다. **막아 주지 않는 것** — 쪼갠 동안 상대 조각을 읽기 전용으로 보기(동결 교차 읽기)와 region 단위 서로소 증명. 짓지 않았고, 짓지 않았다고 적는다.

왜 새 문장 없이 지었나. 언어 설계 문서는 이 문제의 답으로 split region R into R1 … Rn by P 라는 새 문장을 적어 두었다. 그런데 언어에는 이미 둘이 있었다 — **브랜드**(저장소의 정체성을 타입이 든다)와 owned(값이 하나뿐이고 넘기면 손을 떠난다). 둘을 곱하면 토큰이 곧 접근 단위다(19장).

op	모양	실패하면
token · halves	조각 토큰 · 둘로 나눈 결과(low · high)	—
open	comptime b, n u64 → owned token b	없음(범위 0..n)
split_at	comptime b, t owned token b, at u64 → halves b	없음(at 은 범위로 잘린다)
rejoin	comptime b, a owned token b, c owned token b → option (token b)	순서 · 인접이 어긋나면 none
covers	comptime b, t token b, i u64 → bool	—
width	comptime b, t token b → u64	계약(op 이 스스로 적는 약속) 위반은 진입에서 멈춘다
write	comptime b, t, mem mut slice u64, i, v → bool	범위 밖이면 false
read	comptime b, t, mem slice u64, i → option u64	범위 밖이면 none

표 50.1 — shard 의 op

rejoin 의 순서 규약 — 그리고 그것이 계약이 아닌 이유. 합칠 때는 낮은 id 를 먼저 넘긴다(교착 규약: 교차 획득은 언제나 오름차순). 어기면 none 이다. 처음엔 requires lt (field a id) (field c id) . 로 적었는데, 계약 시험 생성기가 두 구조체 인자 사이의 관계에 대해서는 거절 사례를 만들지 못한다. 아무도 검증하지 못하는 계약은 검사가 아니라 문장이므로, 부르는 쪽이 반드시 받는 값으로 내렸다.

> **반례. 거꾸로 합친다**
>
> `shard.rejoin g hi lo` 는 `none` 이다. 멈추지 않으므로 반환값을 보지 않으면 합쳐지지 않은 것을 모른 채 지나간다. `guard is_some back .` 으로 받는다.

> **반례. 브랜드 하나로 저장소 둘을 연다**
>
> 같은 브랜드 `g` 로 `shard.open` 을 두 번 부르면 컴파일 에러 `E-BRAND-REUSED` 다. 브랜드는 저장소 하나의 이름이다. 저장소가 둘이면 `newtype` 도 둘이다(브랜드는 자료를 나르지 않으므로 값이 0 이다).

주의. 토큰은 `var` 에 묶는다(`owned` 는 가변 장소를 요구한다). `split_at` 의 `at` 은 잘린다 — 범위 밖이면 빈 조각이 나오고, 빈 조각은 아무것도 만지지 못하므로 안전하다. 뜨거운 경로에 동기화가 없다 — 토큰이 이미 권한을 말했으므로 남는 것은 범위 검사 하나이고, 시험이 방출된 C 에서 그 경로의 원자 연산 · 락이 0 임을 잰다. 이 모듈은 스레드를 모른다 — 토큰을 실행 단위에 태우는 것은 27장 의 일이다.

budget — 핸들 비트 예산과 세대 한 바퀴

소스 lib/budget.low
층 L1 — 순수 계산
권한 없음

세대 핸들을 **한 워드**에 넣을 때 세 조각에 몇 비트씩 줄지 정하고, 그 예산이 들어맞는지 **컴파일 때** 확인한다. 저장소를 직접 만들 때 — pool · shard 처럼 칸을 나눠 주고 되받는 것을 지을 때 — 핸들의 폭을 소스에 적고 도구가 지키게 하는 자리다.

```
h = (generation << (shard_bits + slot_bits)) ¦ (shard << slot_bits) ¦ slot
      몇 번째 삶                    어느 샤드                  어느 칸
```

```
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 .
```

이 모듈이 지키는 것

들어가지 않는 예산은 컴파일되지 않는다. budget.pack 40 16 16 … 은 합이 72 라 한 워드를 넘는다 — 상수뿐이므로 부르는 자리에서 E-CONTRACT-IMPOSSIBLE 이다. **값이 자기 칸에 들어가지 않으면** none 이다 — 조용히 잘라 담으면 서로 다른 두 핸들이 같아진다. **세대가 한 바퀴 돌면 그 칸은 은퇴한다** — next_gen 이 none 을 답하고 되살리지 않는다. 0 으로 돌아가면 그 칸의 옛 핸들이 전부 되살아나는데, 그것이 세대 핸들이 막으려던 바로 그 일이다.

기계	slot · shard · gen	담는 수	근거
64 비트	32 · 8 · 24	약 43 억 칸 · 256 샤드 · 약 1,700 만 삶	널리 쓰이는 실물 id 중 가장 넓은 것(SQLite 페이지)이 32 비트
32 비트	16 · 4 · 12	65,536 칸 · 16 샤드 · 4,096 삶	lwIP 가 길이를 16 비트로 센다

표 50.1 — 권고 기본값과 근거

기본값일 뿐이다 — 저장소가 자기 수를 적으면 그 수가 이긴다.

op	모양	실패하면
plan • default64 • default32	예산 구조체 · 권고 기본값	—
pow2	n u64 → u64(계약 n < 63)	계약 위반은 진입에서 멈춘다
slots • shards • lives	p plan → option u64	폭이 63 이상이면 none
pack	예산 셋 + 값 셋 → option u64	값이 칸을 넘으면 none · 예산이 넘치면 컴파일 에러
slot_of • shard_of • gen_of	예산 + h → u64	계약 위반은 진입에서 멈춘다
retired	gen_bits, gen → bool	—
next_gen	gen_bits, gen → option u64	은퇴한 칸이면 none

표 50.2 — budget 의 op

반례. 세대를 그냥 올린다

let g u64 be add gen 1 . 은 에러가 없지만, 폭을 넘으면 다른 삶의 핸들과 같아진다. next_gen 을 쓰고 none 을 받는다 — 그 칸은 끝났다는 뜻이다.

> **반례. 63 비트 칸을 센다**
>
> `budget.pow2 63` 은 진입 계약이 거절한다. `shl 1 63` 은 부호 있는 64 비트에서 넘쳐 음수가 된다 — 조용히 틀린 수를 돌려주느니 거절한다.

주의. 되꺼낼 때 **같은 예산**을 준다 — `pack` 과 `slot_of` 가 다른 수를 보면 답이 조용히 틀린다. 예산을 상수 셋으로 한 자리에 적어 두고 그것만 쓴다. op 이름이 `cap` 이 아니라 `pow2` 인 이유 — `cap` 은 문법 자리(`input k cap clock .`)라 op 이름으로 쓰면 `clock` 과 한 단위에 설 수 없었다. 두 워드 핸들은 짓지 않았다.

`wire` — 한 낱말을 칸으로 나눠 쓰기

소스 `lib/wire.low`
층 L1 — 순수 계산
권한 없음

한 정수 안에 여러 값을 나란히 넣어 쓸 때 그 칸을 **마스크 하나로** 다룬다. 마스크가 어디서 시작하는지(자리)와 몇 비트인지(폭)를 둘 다 말하므로, 손으로 적을 때 늘 갈리던 그 두 수가 갈릴 수 없다. 날짜시각을 한 워드로 들고 다닐 때(`clock.local_packed`), 핸들에 샤드 · 칸 · 세대를 넣을 때(`budget`), 장치 레지스터의 필드를 읽고 쓸 때 — 한 낱말을 여러 칸으로 쓰는 모든 자리다.

```
mask = 0x0f00        →  자리 8 · 폭 4
wire.pick  mask w    →  (w & mask) >> 8       그 칸의 값을 꺼낸다
wire.put   mask v    →  (v << 8) & mask       값을 그 칸 자리로 옮긴다
wire.merge mask w v  →  (w & ~mask) ¦ put     낱말의 그 칸만 갈아 끼운다
```

```
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 .
```

이 모듈이 지키는 것

자리와 폭이 한 수에서 나온다 — "자리는 고쳤는데 폭은 안 고쳤다" 가 일어나지 않는다. 손으로 적은 보수 마스크가 2^{32} 만큼 어긋났는데 인자가 0 인 시험이 그것을 통과시킨 기록이 이 모듈의 시험에 남아 있다. **마스크 0 은 거절한다** — 모든 op 이 `requires ne mask 0` 을 든다. **비용은 0 이다** — 상수 마스크의 `trailing_zeros` 는 컴파일 때 접히고, 같은 일을 손으로 쓴 코드와 기계 명령 수가 같다(x86-64 gcc -O2 에서 꺼내기 18 = 18, 갈아 끼우기 27 = 27, 시험이 매번 잰다).

op	하는 일
`shift_of` · `max_of` · `solid`	마스크의 자리 · 칸에 들어가는 최댓값 · 비트가 이어진 마스크인가
`pick` · `put` · `merge`	꺼내기 · 자리로 옮기기 · 그 칸만 갈아 끼우기
`fits`	값이 그 칸에 들어가는가
`polarity v m`	마스크 자리만 뒤집기(active-low)
`reverse_bits v w`	폭 `w` 안에서 비트 차례 뒤집기
`onwire order inv v w`	선 기술자 한 번에(컴파일 때 값)
`permute v tab` · `is_identity tab`	자리 옮김(순열) · 항등인가
`to_set now want` · `to_clear now want`	세울 마스크 · 지울 마스크

표 50.1 — `wire` 의 op

바이트 차례는 빌트인 `byte_swap` 이다. 선으로 나갈 때의 축(극성 · 비트 차례 · 바이트 차례 · 순열)은 서로 독립이다.

왜 "값" 이 아니라 세울 · 지울 짝인가. 포트에 값을 통째로 쓰려면 읽고 → 고치고 → 쓴다. 그 사이에 인터럽트가 다른 자리를 켜면 내 쓰기가 그것을 지워 버린다. 그래서 하드웨어가 세울 마스크 · 지울 마스크 레지스터(STM32 의 `BSRR`)를 준다 — 읽지 않으므로 잃을 것이 없다. 편의가 아니라 정확성이다(30장). **항등은 공짜다** — `onwire 0 0 v w` 는 값을 그냥 돌려주는 코드와 명령 수가 같다. 순열 표는 실행 때 값이라 접히지 않으므로 `is_identity` 로 부르기 전에 물을 수 있게 했다.

> **반례. 마스크와 따로 자리를 또 적는다**
>
> `bit_and (shr w 32) 15` 는 자리(32)와 폭(15)이 두 수라 한쪽만 고치면 조용히 틀린다. `wire.pick` 은 한 수만 받는다.

> **반례. 들어가는지 묻지 않고 넣는다**
>
> 달은 네 비트다. `wire.merge (m_month) w 20` 은 에러 없이 4(= 20 & 15)를 넣는다. `wire.fits` 가 그 물음에 답한다 — 둘을 한 op 으로 묶지 않은 까닭은, 실패를 값으로 답하면 모든 자리에서 `option` 을 풀어야 하기 때문이다.

주의. 마스크는 한 곳에만 적는다(`fn m_month output u64 .` 처럼 이름 하나로). 구멍 난 마스크(비트가 이어지지 않은 것)는 다루지 않는다 — `solid` 가 답한다. 폭 · 프레임 축(8 비트 값을 12 비트 프레임으로)은 없다. op 이름이 `get` · `read` 가 아닌 까닭 — `get` 은 처리기가 이미 쓰는 낱말이라 영원히 호출되지 않았고, `read` 는 C 의 `read` 와 충돌해 네이티브 빌드가 깨졌다. 라이브러리 op 이름은 처리기의 낱말 · 명세의 표 · libc 셋을 모두 피해야 한다.

`flags` — 이름 붙은 켬 · 끔을 한 낱말에

소스 `lib/flags.low`
층 L1 — 순수 계산
권한 없음

켜고 끄는 설정 여럿을 **정수 하나**에 담아 다룬다. 자리는 **비트 번호**(0 번, 1 번, 2 번)로 부른다. 이름은 컴파일할 때만 살고, 실행 때 남는 것은 정수 하나다. 설정 화면의 켬 · 끔 항목, 기능 스위치, 더티 표시, 장치의 상태 비트 — 여럿을 따로 들고 다니기는 아깝고 하나로 묶으면 비교가 쉬운 자리다.

```
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) .
```

이 모듈이 지키는 것

없는 자리는 진입에서 멈춘다 — 자리를 받는 op 은 전부 `requires lt n 64` 를 든다. C 의 `1u << 64` 는 정의되지 않은 동작이고, 여기서는 거절이다. **비용은 0 이다** — 라이브러리로 쓴 코드와 손으로 비트 연산을 쓴 코드의 기계 명령 수가 같다(x86-64 gcc -O2 에서 켜기 17 = 17, 묻기 18 = 18, 부분집합 24 = 24, 시험이 매번 잰다). **자리 번호는 겹치면 안 된다** — 설정을 파일이나 장치에 저장하는 순간 번호는 약속이 된다. 가운데 자리를 지우면 뒤가 밀려 저장된 설정이 조용히 다른 뜻이 되므로, 지운 자리는 비워 두고 재사용하지 않는다.

하는 일	op
빈 값 · 그 자리 하나	`empty` · `of n`
켜기 · 끄기 · 뒤집기 · 묻기	`with s n` · `without s n` · `toggle s n` · `has s n`
합 · 교 · 차 · 부분집합	`join a b` · `both a b` · `minus a b` · `inside a b`
몇 개 · 가장 낮은 자리	`ones s` · `lowest s`(계약 `s > 0`)
하나라도 · 하나도 · 전부	`has_any s` · `has_none s` · `is_full s w`(폭 `w`) · `widthmask w`
무엇이 바뀌었나	`changes a b` · `changes_in a b m`
기본값과 다른 것만 · 이 자리만 갈아 끼우기	`from_default s d` · `overlay base over m`

표 50.1 — `flags` 의 op — 한 낱말(64 자리까지)

64 자리를 넘을 때 는 바이트 배열이고 이름이 다르다 — `wide_bits s`(바이트 수 × 8), `wide_has` · `wide_with` · `wide_without` · `wide_toggle`, `wide_ones`, `wide_next s from`(다음 선 자리, 빈 바이트는 통째로 건너뛴다). 경계의 값은 재서 적었다 — `has s 63` 이 17 명령, `wide_has s 64` 가 26 명령이다. 아홉 명령의 차이는 "어느 바이트인가"(나눗셈 · 색인)와 "그 바이트가 있는가"(경계 검사)의 값이다. 두 구현을 한 이름 뒤에 숨기지 않는 까닭이다. 범위 밖은 값으로 답한다(`false` · `0`) — 폭이 실행 때 정해지므로 계약으로 적을 수 없다.

이름은 옵트인이다. 이름표는 부르는 쪽의 것이고 라이브러리는 문자열을 하나도 들지 않는다. `flags.label_in "sound\x00music\x00shadow\x00" 2` 는 `"shadow"`, `label_count` 는 이름 수를 답한다. 이름을 쓰지 않는 프로그램의 바이너리에는 그 글자가 없다 — 시험이 두 방출물의 읽기 전용 영역 크기를 비교한다.

여럿이 같은 낱말을 고칠 때. 이 모듈의 op 은 값을 받아 값을 돌려준다 — 저장된 낱말을 직접 고치지 않는다. 여러 실행 흐름이 `set shared (flags.with shared 2)` 로 고치면 읽고 → 고치고 → 쓰기 사이에 남의 갱신이 사라진다. 그 자리는 원자적 연산이 답한다 — `atomic_or s 0 4`(2 번 켜기), `atomic_and s 0 251`(끄기), `atomic_xor s 0 4`(뒤집기). 자리가 상수면 마스크도 상수라 기계 명령 하나로 내려간다(27장). 장치 레지스터라면 `wire` 의 `to_set` · `to_clear` 를 쓴다.

> **반례. 자리를 마스크로 적는다**
>
> `flags.with (flags.empty) 64` 는 마스크 64(= 6 번 자리)를 뜻했지만 이 모듈은 "64 번 자리" 로 읽고, 양쪽이 상수라 컴파일 에러(`E-CONTRACT-IMPOSSIBLE`)다. 여기서 수는 언제나 번호다. 마스크로 다루려면 `wire` 를 쓴다.

> **반례. 설정을 지금 값으로만 저장한다**
>
> 기본값이 바뀌면 저장된 값이 낡는다. `flags.from_default s d` 로 차이만 적으면 작고, 기본값이 바뀌어도 뜻이 산다.

주의. 폭이 1 을 넘는 필드는 `wire` 의 일이다. 자리 번호의 자동 관리는 없다 — 번호는 사람이 적고, 두 이름이 같은 자리를 쓰면 개발 저장소의 검사가 잡는다. op 이름이 `set` · `union` · `count` · `first` 가 아닌 까닭 — 문법 낱말이거나 이미 있는 연산이거나, 다른 파일이 지역 이름으로 쓰고 있었다. `first` 를 내보내자 `lib/regex.low` 가 깨졌고 조합 검사가 그것을 잡았다.

segarena — 고정 크기 세그먼트 아레나

소스 lib/segarena.low
층 L1 — 호출자의 저장
권한 없음

저장소를 한 덩어리로 잡지 않고 **같은 크기의 조각(segment)** 으로 늘려 간다. 조각 안은 연속이라 훑기가 빠르고, 자라날 때 이미 있는 것을 옮기지 않는다. 색인 하나가 조각과 자리로 갈린다 — 그게 전부다.

```
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 .
```

비용이 op 이름에 적혀 있다 — 이것이 이 모듈의 요점이다

같은 합을 2,048 만 원소에서 두 방법으로 쟀다. 조각을 한 번 빌리는 sum_seg 는 조각마다 directory 를 한 번 보고 21 ms, 전역 색인으로 매번 읽는 sum_global 은 원소마다 보고 81 ms 였다. 답은 같다. 컴파일러는 세그먼트를 모른다 — 빠른 쪽을 고르는 것은 **당신이 부르는 op** 이다. 전역 임의 접근의 비용을 평평한 slice 처럼 숨기지 않는다.

op	하는 일	실패하면
arena	아레나 타입(커서와 모양만 든다)	—
open	아레나를 연다(조각당 2^kbits 칸)	계약 1 ≤ kbits ≤ 20 — 상수면 컴파일 때 거절
grow	조각을 하나 더 열고 dir 에 시작 자리를 적는다	자리가 없거나 용량을 넘으면 none
seg_slots · seg_of · slot_of	조각당 칸 수 · 전역 색인 → 조각 · 자리	—
at	전역 색인으로 한 칸 읽기(편하다, directory 를 매번 본다)	범위 밖 · 열지 않은 조각이면 none
walk_base	조각의 시작 자리를 한 번 빌린다(뜨거운 순회용)	열지 않은 조각이면 none
sum_seg · sum_global	같은 합을 두 경로로 — 비용 대조의 증인(주장이 참임을 보여 주는 자리)	—

표 50.1 — segarena 의 op

반례. 조각 크기가 뜻이 없다

segarena.open g 0 은 컴파일 에러 E-CONTRACT-IMPOSSIBLE 이다 — 조각이 한 칸이면 세그먼트가 아니다. 인자가 상수라 부르는 자리에서 판정된다(14장).

반례. 뜨거운 순회를 전역 색인으로 돈다

while 안에서 원소마다 segarena.at 을 부르면 에러는 없고 느릴 뿐이다(측정에서 3.9 배). 조각 단위로 돌면서 walk_base 로 시작 자리를 한 번만 빌린다.

주의. 저장은 호출자 것이다 — `dir`(조각마다 시작 자리)과 `mem`(칸이 사는 곳)을 밖에서 준다(`pool` · `shard`와 같은 규율). 회수는 일괄이다 — 조각 하나만 돌려주는 길은 없다. 개별 회수가 필요하면 `pool` 이 그 자리다. 가변 크기 조각은 짓지 않는다 — `i >> k` 공식이 서지 않고 비용이 달라지므로 다른 모듈이어야 한다. 조각을 빌리는 뷰는 `segview` 가 맡는다.

pagecache — 페이지 id 와 고정 커서

소스 lib/pagecache.low
층 L1 — 호출자의 저장
권한 없음

페이지를 담아 두는 캐시에서 **오래 사는 이름**(페이지 id)과 **짧게 닿는 권한**(pin)을 가른다. 밖에 저장하는 것은 언제나 id 이고, 바이트를 만지는 동안만 pin 을 든다. SQLite 의 페이지 캐시가 그 모양이다. 페이지는 옮겨진다(축출 · 재배치). 옮겨질 때 고칠 것이 id → 자리 표 하나뿐이어야지, 흩어진 포인터를 찾아다녀야 한다면 그 캐시는 쓸 수 없다.

```
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 이 사이에는 evict 를 부를 수 없다 --- 컴파일되지 않는다
var c2 owned pagecache.cache db . be pagecache.release db p .
var c3 owned pagecache.cache db . be pagecache.evict db c2 .
```

고정이 살아 있는 동안 축출은 컴파일 에러다

> acquire 는 캐시 토큰을 **삼킨다**. 그래서 pin 이 사는 동안에는 evict · reset 에 건넬 것이 손에 없다 — E-OWN-MOVED. 그것을 막는 것은 이 모듈이 아니라 언어다. owned 가 "토큰은 하나이고 넘기면 손을 떠난다"를 이미 강제한다(19장). 브랜드로 저장소를 봉하고, 토큰으로 접근 단위를 가르고 (shard), 여기서 세 번째로 같은 규칙이 새 규약을 공짜로 만든다.

무엇	정한 것	근거
기본 pin 상한	machine.cache_line / 8 — x86_64 · arm64 8 · mips_be 4 · cortex_m 1	PostgreSQL 이 pin 표 8 칸을 "캐시 라인 크기쯤인 64 바이트" 로 두었다
release 순서	LIFO — 겹친 pin 은 앞선 것을 삼킨다	SQLite 커서가 페이지를 스택으로 든다

표 50.1 — 상한과 순서 — 그 수가 어디서 왔나

수 대신 이유를 적었다. 8 을 박아 두면 캐시 라인이 다른 기계에서 틀린다 — 그래서 machine.cache_line 으로 나눈다(30장).

op	하는 일	실패하면
cache • pin • pin2	캐시 토큰 · 고정 · 겹친 고정 타입	—
open	캐시 토큰을 만든다	—
acquire	페이지를 고정한다 — 캐시 토큰을 삼킨다	—
acquire_more	겹쳐 고정한다 — 앞선 pin 을 삼킨다(깊이 +1)	—
release • release_more	놓는다 — 토큰을 돌려준다	—
evict • reset	축출 · 비움. 캐시 토큰을 요구한다	고정 중이면 컴파일 에러
slot_of	id 의 현재 자리	빈 캐시면 none
pin_limit • within_limit	이 기계의 상한과 그 판정	—
epoch_of • depth_of • depth2_of	관측	—

표 50.2 — pagecache 의 op

> **반례. 겹친 고정을 안쪽부터 놓는다**
>
> `a` 를 고정하고 `acquire_more db a 9` 로 `b` 를 겹친 뒤 `release db a` 를 부르면 컴파일 에러다 — `a` 는 `b` 안에 있다. 순서를 어길 길이 없다. 그것이 LIFO 다.

> **반례. 자리를 저장해 두고 나중에 쓴다**
>
> `slot_of` 로 얻은 자리를 들고 있다가 축출이 지나간 뒤 쓰면 에러는 없고 틀린 자리를 읽을 뿐이다. 밖에 저장하는 것은 id 여야 한다 — 자리는 물을 때마다 새로 얻는다. 그것이 이 모듈이 id 와 자리를 가른 이유다.

주의. 자리도 pin 도 저장하지 않는다 — 둘 다 지금 이 순간의 것이다. 상한은 규약이지 강제가 아니다(깊이는 `pin2` 까지만 지었다). 페이지를 읽어 오는 일과 그 실패, 그리고 LRU 같은 축출 정책은 여기 없다 — 정책은 이 규약 위에 얹는다.

growvec — 자가성장 바이트 벡터

소스 lib/growvec.low
층 L1 — 저장(얼로케이터를 들고 다닌다)
권한 없음 — open 이 범프 얼로케이터를 건네받는다

자리가 모자라면 알아서 더 받아 오는 **바이트** 배열이다. 파일을 다 읽을 때까지, 사용자가 Enter 를 누를 때까지처럼 얼마나 들어올지 모를 때 쓴다. growvec.gvec 은 vecgen.vec u8 allocs.bump_bytes 의 **별칭**이고 op 들은 vecgen 으로 위임한다. 바이트만 밀 때는 타입 인자를 적지 않아도 되는 이쪽이 짧다 — 이름이 짧은 것도 인터페이스다.

```
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 .
```

왜 있는가. vecs 로 바이트를 모으면 부를 때마다 얼로케이터 · 버퍼 · 길이를 넘기고, 늘어난 새 버퍼를 받아 **다시 묶는 일**도 호출부의 몫이다. 그 한 줄을 잊어도 컴파일러가 잡지 못한다 — 옛 버퍼도 유효한 슬라이스이기 때문이다. growvec 은 셋을 **한 값 안에** 넣어 그런 결함을 구조로 없앤다. 그래도 마법은 없다 — 얼로케이터는 여전히 누군가 넣어 준 것이고, 성장의 비용은 false 를 돌려줄 수 있는 op 자리에 보인다.

op	effects	하는 일
gvec	—	vecgen.vec u8 allocs.bump_bytes 의 별칭
open cap0(using al)	state	빈 벡터를 연다. cap0 은 0 도 된다(첫 push 에서 8 로). 못 받으면 none
count_of g · cap_of g	none	담긴 개수 · 지금 용량
add_byte g b	state	바이트 하나 — 모자라면 두 배 + 8 로 늘린다. false = 새 자리를 못 받음(벡터는 여전히 유효)
add_all g s	state	바이트열 통째로. **중간에 실패하면 앞부분은 들어가 있다**
reserve_more g more	state	n + more 가 들어갈 자리를 미리 확보
view_of g	none	담긴 만큼만 보는 뷰

표 50.1 — growvec 의 op

add_all 은 전량 아니면 무를 약속하지 않는다. 약속하려면 먼저 자리를 확보해야 하고, 그 비용을 낼지는 호출자가 고를 일이라 reserve_more 를 따로 뒀다.

```
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
```

메모리 쪽의 값. 범프 얼로케이터는 되돌리지 않으므로, 한때 벡터가 자랄 때마다 옛 버퍼가 아레나에 버려진 채 남아 2,048 바이트를 담는 데 8,104 바이트를 썼다. 얼로케이터에 **제자리 성장**(grow — 마지막 할당이 우리 버퍼면 그 자리를 늘린다)을 두어 그 세대를 없앴다. 남은 두 배는 성장 정책의 여유(용량 ≈ 2n)이고, 시간과의 정상적인 교환이다. 넣을 크기를 미리 알면 reserve_more 로 여유까지 없애 정확히 필요한 만큼만 잡는다(allocs).

> **반례. 성장 전에 꺼낸 뷰를 성장 뒤에 쓴다**
>
> `view_of` 로 꺼낸 뒤 `add_all` 을 부르면 버퍼가 바뀔 수 있고, 뷰는 옛 버퍼를 본다. 늘어나지 않았다면 우연히 맞아서 더 나쁘다. 뷰는 마지막에 꺼낸다.

> **반례. `gvec` 을 복사해 두고 양쪽에 넣는다**
>
> `var b growvec.gvec be a .` 뒤 `a` 와 `b` 에 각각 넣으면 둘이 같은 버퍼를 보면서 `n` 은 서로의 변경을 모른다 — 서로 덮어쓴다. 한 벡터는 한 이름으로 다루고, 넘길 때는 `mut gvec` 로 넘긴다.

> **반례. 반환값을 보지 않는다 · `cap_of` 만큼 읽는다**
>
> `add_byte` 의 `false` 를 보지 않으면 벡터는 그 바이트 없이 계속 간다. `n` 뒤는 쓰레기다 — `count_of` 나 `view_of` 를 쓴다.

주의. 개별 반납과 축소는 없다. 벡터의 수명은 얼로케이터의 수명 안이다. `effects state` 라 순수 계층에서는 쓸 수 없다 — 순수하게 풀 수 있는 문제라면 호출자 버퍼 (`fmt` 의 규약)가 더 싸다.

vecgen — 제네릭 자가성장 벡터 `vec t a`

소스 `lib/vecgen.low`
층 L1 — 저장(얼로케이터를 들고 다닌다)
권한 없음 — `open` 이 얼로케이터를 건네받는다

어떤 타입이든 담고, 자리가 모자라면 스스로 커지는 배열이다. 축이 둘이다 — **원소 타입** `t` × **얼로케이터 타입** `a`. 같은 소스에서 `vec#u32#allocs.bump_bytes` 와 `vec#u32#allocs.bump_aligned` 같은 인스턴스가 나오고(`--ir` 에 보인다), vtable 도 간접 호출도 없다 — 정책이 컨테이너 밖에 있고 고르는 값이 0 이다 (22장, 34장).

```
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 .
```

타입을 매번 적는다. 추론이 없어서가 아니라 일부러 넣지 않았다 — 어떤 인스턴스가 생겼는지(바이너리에 코드가 몇 벌 생기는지) 소스에 보이고, 읽는 사람이 `v` 의 원소 타입을 선언까지 되짚어 가지 않아도 안다. 추론은 나중에 덧붙일 수 있지만 그 반대는 어렵다. 바이트만 쓴다면 타입 인자를 하나도 적지 않는 `growvec` 이 짧다.

설계. ① **단형화다** — `vec u32` 는 컴파일 때 구체 구조체가 되고, 크기와 오프셋이 전부 상수로 나온다. 대가는 인스턴스마다 코드가 생기는 것이고, 그것은 감춘 비용이 아니라 보이는 비용이다. ② **저장은 바이트다** — 얼로케이터가 바이트를 주기 때문이다. 원소로 보는 것은 `view_array` 가 하고, 몇 바이트가 필요한지는 `size_of t` 가 답한다. 이것이 없으면 제네릭 코드는 최대 폭(8 바이트)으로 잡아 `vec u8` 이 8 배를 쓴다. ③ **권한은 남이 준 것이다** — 컨테이너가 권한을 만들지 않고 들고 다닐 뿐이다.

op	effects	하는 일
`vec`	—	구조체 — `al a · store mut slice u8 · n u64`(담긴 **원소** 수)
`open t cap0(using al)`	state via a	빈 벡터를 연다(원소 `cap0` 개 자리). 첫 자리를 못 받으면 `none`
`count_of t a g · cap_of t a g`	none	담긴 원소 수 · 지금 용량(원소 수)
`append t a g x`	state via a	원소 하나 — 모자라면 두 배 + 8 원소로 커진다. `false` = 새 자리를 못 받음
`at t a g i`	none	i 번째 원소 → `option t`
`set_at t a g i x`	none	i 번째를 고친다. 범위 밖이면 `false` — 조용히 늘리지 않는다
`reserve_more t a g n`	state via a	앞으로 n 개 더 들어갈 자리를 미리
`view_of t a g`	none	담긴 만큼만 보는 `slice t`

표 50.1 — `vecgen` 의 op — 첫 두 인자가 원소 타입 · 얼로케이터 타입(`open` 은 원소 타입만)

세는 단위를 섞지 않는다 — `cap0` · `cap_of` · `count_of` · `at` · `set_at` 은 전부 **원소 개수**이고 바이트는 모듈 밖으로 나오지 않는다. 원소는 크기 있는 스칼라이거나 **viewable 구조체**(모든 필드에 바이트 레이아웃이 있고 총 255 바이트 이하)다 — `at` 이 그 구조체를 필드째 돌려준다. `view_of` 의 결과는 그대로 `sortgen` 이나 `searchlib` 에 넘길 수 있다.

미리 자리를 잡으면 "전량 아니면 무" 가 된다. `reserve_more` 가 성공한 뒤 이어지는 `append` 는 도중에 실패하지 않는다.

```
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
```

> **반례. 커진 뒤에 옛 뷰를 쓴다**
>
> `view_of` 로 꺼낸 `s` 를 들고 `append` 를 부르면 저장소가 바뀔 수 있고, `s` 는 옛 버퍼를 본다. 커지지 않았다면 우연히 맞아서 더 나쁘다. 뷰는 마지막에 꺼낸다.

> **반례. `append` 의 `false` 를 보지 않는다 · `cap_of` 만큼 읽는다**
>
> `false` 는 OOM 이고, 보지 않으면 조용히 짧아진 벡터가 된다. 용량 뒤는 쓰레기다 — `count_of` 나 `view_of` 를 쓴다.

> **반례. 레이아웃 없는 구조체 · 중첩**
>
> 슬라이스 필드가 있는 구조체로 `open` 하면 바이트 레이아웃이 없어 배열도 없다(`E-IR-UNDEF`). `vec (vec u8 …)` 같은 중첩도 없다 — 타입 인자는 낱말 하나다.

주의. 개별 반납과 축소는 없다 — 범프 얼로케이터는 통째로만 되돌리므로 성장 횟수를 줄이려면 `cap0` 이나 `reserve_more` 를 쓴다. 벡터의 수명은 얼로케이터의 수명 안이다. 소유 있는 원소(`vec files.handle`)는 지금 인스턴스 안에서 걸리고 진단이 호출 자리를 가리키지 않는다 — 남은 결함이다. 버퍼를 호출자가 드는 컨테이너는 `vecs` 로 따로 있다 — 소유 모델이 달라서 접지 않았다.

mapgen — 제네릭 해시맵 table k v

소스 lib/mapgen.low
층 L1 — 저장(얼로케이터를 들고 다닌다)
권한 없음 — open 이 범프 얼로케이터를 건네받는다

키로 값을 찾는 표이고, 키 타입과 값 타입을 **내가 고른다**. op 을 부를 때마다 키 타입과 값 타입을 이 순서로 적는다(vecgen 과 같은 이유다). hashmap 은 u64 → u64 로 못 박혀 있고 저장소를 호출자가 들며 rehash 도 직접 부른다. mapgen 은 타입이 열려 있고, 저장소와 얼로케이터를 한 값이 소유하며, 꽉 차면 스스로 다시 뿌린다.

```
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 .
```

설계. ⓪ 이름이 map 이 아니라 table 인 이유 — map 은 빌트인 op 이다. 같은 낱말이 두 가지를 뜻하면 문장이 어떻게 묶이는지가 흔들린다. 제네릭이라는 이유로 그 검사를 빠져나가던 구멍을 닫자 이 모듈이 첫 번째로 걸렸다. ① **빈칸을 키 값으로 예약하지 않는다** — 상태 배열(flags: 0 빈칸 · 1 찬 자리 · 2 무덤)을 따로 둔다. "키 0 = 빈칸" 이라는 흔한 지름길은 키 0 을 담긴 척하며 조용히 틀린 답을 준다. ② **무덤을 남긴다** — 지운 자리를 빈칸으로 되돌리면 그 자리를 지나 놓인 키를 영영 못 찾는다. ③ **부하율 ≤ 0.5** — 절반을 넘기 전에 슬롯을 두 배로 늘리고 전부 다시 뿌린다. ④ 해시는 분포용이지 암호용이 아니다(곱셈-xorshift 한 판). ⑤ **순회는 반복자가 아니라 자리다** — 해시 표에는 "다음 원소" 라는 자연스러운 순서가 없으므로, 상태를 든 반복자 객체 대신 next_used 가 다음으로 찬 자리를 준다. 무효화될 객체가 없다.

op	effects	하는 일
table	—	구조체 — al · keys · vals · flags(바이트) · n(담긴 쌍 수)
open k v want(using al)	state	빈 맵을 연다(쌍 want 개 준비, 슬롯은 (want+1)*2 를 2 의 거듭제곱으로, 최소 8)
insert k v m key val	state	넣는다(있으면 덮어쓴다). 부하율을 넘게 되면 먼저 다시 뿌린다. false = 자리 부족
lookup k v m key • has k v m key	none	값 → option v · 있는가
erase k v m key	none	지운다(무덤을 남긴다). 있었으면 true
count_of k v m • slots_of k v m	none	담긴 쌍 수(무덤은 세지 않는다) · 슬롯 수
next_used k v m from	none	from 부터 처음 찬 자리 → option u64
key_at k v m at • val_at k v m at	none	그 자리의 키 · 값. 찬 자리가 아니면 none

표 50.1 — mapgen 의 op — 첫 두 인자가 키 타입 · 값 타입

키 배열은 슬롯 × size_of k 바이트, 값 배열은 슬롯 × size_of v 바이트다 — table u8 u16 을 3 쌍으로 열면 슬롯 8 에 32 바이트, 같은 개수를 table u32 u64 로 열면 104 바이트다. 산술이 인스턴스마다 다르다.

```
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
```

빈도 세기의 관용구는 `lookup` → 없으면 1, 있으면 +1 → `insert` 다. 값 타입을 `u8` 로 두고 1 만 넣으면 집합이다.

> **반례. 순회하면서 넣는다**
>
> `insert` 가 다시 뿌리면 자리 번호의 뜻이 바뀌어 건너뛰거나 두 번 본다. 넣을 것을 모아 두었다가 순회가 끝난 뒤 넣는다. `erase` 는 순회 중에도 안전하다(무덤만 남는다).

> **반례. `at` 을 1 씩 올리지 않는다**
>
> `set at (some_value nx)` 로 두면 같은 자리를 영원히 다시 찾는다. 다음 시작은 **앞 자리** + **1** 이다.

> **반례. 정수 아닌 키 · 타입 순서 바꾸기**
>
> `table f32 u64` 는 `widen u64` 가 뜻 없는 비트를 만들어 쓸 수 없다 — 바이트열 키는 `strmap` 이다. `table u32 u64` 를 열고 `u64 u32` 로 부르면 다른 인스턴스를 부르는 것이다(타입 검사가 잡는다).

주의. 키 · 값은 정수 스칼라다. `lookup` 의 `none` 은 에러가 아니라 "그 키가 없다" 는 흔한 정상 경우다. `erase` 는 `count_of` 를 줄이지만 슬롯은 줄지 않는다. 순서는 약속하지 않는다(해시 순서이고 다시 뿌리면 바뀐다). 성장은 통째로 다시 뿌리므로 그 순간 O(슬롯)이 들고, 범프 위에서는 옛 배열 셋이 남는다 — `want` 를 제대로 주는 것이 실질적인 이득이다. 맵의 수명은 얼로케이터의 수명 안이다.

nodelist — 고정 intrusive 목록

소스 lib/nodelist.low
층 L1 — 호출자의 저장
권한 없음

노드가 링크를 **자기 안에** 든다(intrusive). 목록 쪽에 따로 칸을 잡지 않으므로 넣고 빼는 데 할당이 없다 — Linux 커널의 list_head 가 그 모양이다. 위험한 것은 둘뿐이다 — 링크된 노드를 **옮기면** 이웃이 가리키던 자리가 썩고, 링크된 채 **회수하면** 목록이 썩는다.

그 둘을 언어가 막는다. 노드에는 **이동권** (site)이 있고, 목록에 넣으면 목록이 그것을 먹는다(held). 링크된 채 옮기려는 프로그램은 건넬 값이 없어 컴파일이 거절한다.

```
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 는 이미 먹혔다)
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	하는 일
site · held	이동권(링크 안 됨) · 링크된 표 — 브랜드 붙은 타입
nl_open · nl_relocate	자리를 이동권으로 연다 · 링크 안 된 노드를 옮긴다
nl_link · nl_unlink	이동권을 먹고 링크 표를 낸다 · 표를 먹고 이동권을 돌려준다
nl_slot_of · nl_held_slot	이동권 · 링크 표의 자리 번호
nl_solo · nl_after · nl_cut	혼자인 고리로 · at 뒤에 넣기 · 빼기(nx · pv 를 고친다)
nl_walk_cut	머리부터 돌면서 한 노드를 뺀다(가변 순회)
nl_count	고리의 노드 수

표 50.1 — nodelist 의 op

목록은 고리다. 널이 없으므로 끝을 표시할 값이 없다. 대신 머리로 돌아오면 한 바퀴다. 혼자인 노드는 자기 자신을 가리키는 고리 하나다. 처음엔 "자기 자신을 가리키면 끝" 으로 셌다가 틀렸다 — 둘 이상인 고리에서는 아무도 자기를 가리키지 않으므로 그 조건이 영영 오지 않는다.

순회하면서 뺄 때는 다음을 먼저 읽는다. 지금 노드를 빼면 그 next 가 자기 자신이 되므로, 빼기 전에 읽어 두지 않으면 순회가 그 자리에서 멈춘다. nl_walk_cut 이 그렇게 한다.

저장은 호출자 것이다. nx(다음) · pv(이전) 두 슬라이스를 밖에서 받는다. 링크는 **인덱스**다 — 생 포인터가 없고, 인덱스는 segarena 처럼 움직이지 않는 저장과 맞물린다.

여러 목록에 동시에 드는 노드. LRU 목록과 해시 버킷에 같은 노드가 든다면 이동권이 하나뿐이라 두 번째 링크가 들지 못한다. shard 토큰으로 쪼갠다 — role 마다 조각 하나, 전부 되합쳐야(rejoin) 이동권이 다시 선다. 그것이 곧 "모든 role 에서 빼고 나서 회수" 다. 브랜드 하나는 저장소 하나를 이름하므로, 노드 자리와 role 토큰에 같은 브랜드를 쓰면 E-BRAND-REUSED 가 거절한다 — 이 매뉴얼을 쓰다 실제로 걸렸고 컴파일러가 옳았다.

짓지 않은 것 — 자동 회수(첫 회수 정책은 일괄이다), 순환 검출, 동시 순회, 링크된 노드 재배치(고정이 그 반대다).

segview — 조각 뷰의 커서 · 총길이 · 펴기

소스 lib/segview.low
층 L1 — 호출자의 저장
권한 없음

언어가 든 조각 뷰 연산은 셋뿐이다 — view_segments(짓기), segs(조각 수), seg(i 번째 조각). 이 모듈은 그 위에 **커서 · 총길이 · 펴기(coalesce)** 를 얹는다. 셋 다 앞의 셋으로 쓰이므로 내장 연산이 될 이유가 없다.

왜 segments 가 언어에 있나. 조각 여럿을 한 값으로 들지 못하면 밖으로 내보낼 때 조각마다 한 번씩 나가야 한다. 64 KiB 를 64 조각으로 내보낼 때 그 차이가 개발 저장소의 측정에서 28 배였다(writev 한 번 대 write 64 번). 조각은 곧 iovec 이다 — 방출 C 의 슬라이스가 struct iovec 과 크기도 두 오프셋도 같아서, extern 경계에서 조각 배열을 넘기는 값은 캐스트이지 복사가 아니다(시험이 매번 대조한다).

```
rem 서술자는 평평한 slice u64 에 (at, n) 쌍 --- 새 구조체 타입은 없다
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 .
```

부르는 것	무엇을 만지나	값
segs · seg(내장)	서술자만	조각당 수 ns 이하, 복사 0
segview.total_len	서술자만	O(조각 수)
segview.cursor · seek · byte_at	서술자만	O(조각 수) — 조각을 훑어 자리를 찾는다
segview.coalesce	**바이트 전부**	O(n) — 여기서만 복사가 일어난다

표 50.1 — 비용이 op 이름에 적혀 있다

그것이 설계의 요점이다 — 순회는 조각을 빌리기만 하고, 한 덩어리로 펴는 값은 부를 때만 치른다. 비용이 부르는 자리에 보인다.

알아 둘 것. 저장은 호출자 것이다 — coalesce 는 받은 버퍼에 쓰고 **쓴 길이**를 답하며, 버퍼가 모자라면 잘라 쓰지 않고 none 이다. 없는 자리를 가리키는 커서를 만들지 않는다 — seek 이 끝을 넘으면 none 이다.

이름 하나가 남의 모듈을 아프게 했다. 처음에 total_len 의 이름은 total 이었다. 그러자 라이브러리 짝시험이 segview 와 tls13 · tlssrv 를 함께 쓸 때 깨지는 것을 잡았다 — tls13 안의 지역 변수 total 과 부딪혀 엉뚱한 진단(E-GUARD-FALLTHROUGH)이 났다. 흔한 낱말을 export 하지 않는다. (진단이 이름 충돌을 이름 충돌로 말하지 않은 것은 따로 남은 결함이다.)

짓지 않은 것 — 가변 원소 폭(segments T 의 T 는 오늘 스칼라 슬라이스다), coalesce 의 무복사 최적화, 읽기 쪽(readv) 짝.

lifemode · lifeatom — 값이 언제 끝나는가

소스 lib/lifemode.low · lib/lifeatom.low
층 L1 — 호출자의 저장
권한 lifeatom 에 cap atomic

값의 끝을 정하는 방법은 넷이고, 넷 다 값이 다르다. 새 낱말도 새 내장 연산도 늘지 않았다 — 언어가 이미 알던 것을 쓰기 좋게 묶을 뿐이다.

모드	무엇으로 쓰나	값
① 주인이 하나	owned — 낱말 하나면 끝이다(19장)	셈이 없다
② 스레드에 국한된 참조 셈	lifemode.near_* — 그냥 더하기	원자 명령 0
③ 원자적으로 공유하는 참조 셈	lifeatom.rc_* + cap atomic	원자 명령이 있다
④ 바깥의 완료	완료권을 owned 표로	셈이 없다, 대신 컴파일러가 센다

표 50.1 — 값의 끝을 정하는 네 방법

```
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 0 을 답하면 마지막이었다 --- 그때 부른 쪽이 저장을 거둔다
```

원자 쪽은 모듈이 다르다 — lifeatom.rc_open · rc_share k c · rc_release k c(k = cap atomic). rc_release 가 0 을 답하면 마지막이었다. ④ 는 라이브러리가 아니라 모양이다 — fn finish input t owned mymod.ticket . output result u64 mymod.late . 처럼 완료가 완료권을 **먹게** 하면, 두 번 완료하면 E-OWN-MOVED, 완료를 잊으면(완료가 실패할 수 있을 때) E-OWN-INCOMPLETE 다. **정확히 한 번** 이 라이브러리가 아니라 언어에서 나온다.

op	하는 일
lifemode.ceiling	두 모드가 함께 보는 셈의 상한
lifemode.near_open · near_share · near_drop	국한 셈: 1 로 열기 · 하나 늘리기 · 하나 줄이기(0 = 마지막)
lifeatom.rc_open · rc_share · rc_release	원자 셈: 같은 셋, cap atomic 을 받는다

표 50.2 — lifemode · lifeatom 의 op

왜 모듈이 둘인가 — use 는 부르는 것이 아니라 내보내는 것만큼 치른다. 처음에는 넷을 한 모듈에 넣었다. 그랬더니 국한 셈만 쓰는 프로그램의 바이너리에 lock 접두 명령이 2 개 남았다. export 인 op 은 C ABI 래퍼로 전역 심볼을 얻으므로 링커가 버리지 못한다. 원자를 lifeatom 으로 가르자 0 개가 됐다. 원자를 들이는 값이 use lifeatom 한 줄에 보인다.

넘침 — 포화하고, 그 객체는 영원히 산다. 그냥 원자 덧셈으로 세면 상한에서 조용히 0 으로 감긴다(네이티브 실행으로 확인했다). 감긴 수는 이르게 0 에 닿아 살아 있는 객체를 해제한다 — 해제 후 사용으로 가는 길이다. 그래서 이 모듈은 **포화한다**(Linux refcount_t 가 고른 답과 같다). 상한에 닿으면 더 늘리지도 줄이지도 않는다. 그 객체는 죽지 않지만 메모리 안전은 깨지지 않는다. 값은 명령 수 약 19 % 증가이고, 원자 쪽 lock 수는 시도당 하나로 같다(CAS 고리라 경합할 때만 돈다).

짓지 않은 것 — 진짜 비동기 완료(뒷받침 없는 표면은 만들지 않는다), 완료 콜백(호출 시점과 효과를 감춘다), 약한 참조, 순환 수집.

io — 슬라이스 위의 스트림 읽기

소스 lib/io.low
층 L2 — 바깥 세계의 순수한 절반
권한 없음 — 바이트를 채워 오는 쪽만 권한이 든다

메모리에 이미 들어온 바이트열을 **스트림처럼** 조금씩 꺼내 읽는다. 파일을 통째로 읽어 놓고 줄 단위로 훑을 때, 파서에 먹일 커서가 필요할 때 쓴다. 읽기 표면이 `read(buf) → n` 이 아니라 **peek · take · toss → 뷰**다 — 호출자 메모리를 채우는 대신 원본의 한 구간을 가리키는 빌린 뷰를 건넨다(복사 0).

```
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 .
```

왜 권한이 필요 없나. 이 모듈은 이미 메모리에 있는 바이트를 자르기만 하고 커널에 닿지 않는다. 바이트를 실제로 채워 오는 쪽(`read_in` 은 `cap io`, `files.read` 는 `cap file_system`)만 권한을 요구한다. 그래서 프레이밍 로직 전부(버퍼 관리 · `take_line`)를 커널을 건드리기 전에 VM 과 네이티브의 대조로 검증해 두고, 커널 읽기는 이 검증된 로직에 바이트를 공급하는 얇은 잎으로 남는다(28장).

none 은 실패가 아니다

> `peek · take · take_until · take_line` 의 `none` 은 **부재**다 — "청한 만큼의 바이트가 남아 있지 않다". `mem_reader` 는 전부 산술이라 실패할 자리가 없다. 그래서 `files` 가 `result (option T) file_error` 세 자리로 옮겨 갈 때 이 모듈은 따라가지 않았다 — 여기에 `result` 를 붙이면 절대 생기지 않는 `error` 가지를 부르는 쪽마다 만들게 된다. 세 자리가 필요한 것은 fd 를 직접 읽는 리더이고, 그것은 아직 없다.

op	모양 · effects	실패
`scan_until`	`fn (src slice u8, from u64, delim u8, limit u64) → option u64`	limit 안에 없거나 끝이면 `none`
`mem_reader`	actor — state `src slice u8 · pos u64`	—
`attach s`	→ `u64`(언제나 0), state — 소스를 걸고 커서 0. 다시 부르면 처음부터	없음
`remaining`	→ `u64`, none	없음
`peek n`	→ `option slice u8`, none — 소비하지 않고 본다	남은 게 n 미만이면 `none`
`take n`	→ `option slice u8`, state	모자라면 `none`, 커서 불변
`take_rest`	→ `slice u8`, state	없음(끝이면 빈 뷰)
`toss n`	→ `u64`(실제 버린 수), state	없음(끝까지만 버린다)
`take_until delim limit`	→ `option slice u8`, state — 구분자는 소비하되 뷰에 넣지 않는다	못 찾으면 `none`, 커서 불변
`take_line limit`	→ `option slice u8`, state — `take_until 10 limit`	LF 를 limit 안에 못 찾으면 `none`

표 50.1 — `io` 의 op

설계. EOF 는 오류가 아니라 `none` 이다. **상한(limit)이 필수 인자다** — 기본값이 없어 적대적 입력이 무한 읽기를 유발할 수 없고, 한 줄이 정당하게 길어야 얼마인지는 그 데이터를 아는 호출자만 안다. **부분 소비가 없다** — 못 주면 `none` 이고 커서를 움직이지 않으므로 다른 길이로 다시 시도해도 안전하다. **개행 규약은 LF 만** — CR 은 내용에 남는다. 라이브러리가 조용히 바이트를 지우지 않는다(지우면 되돌릴 수 없고, 안 지우면 호출자가 고를 수 있다). 위치 찾기는 순수 op `scan_until` 이 원시 연산이고 상태 있는 커

서(actor)는 그 위의 얇은 껍질이다 — 커서를 값으로 들고 싶으면(재진입 · 역추적) `scan_until` 과 호출자 pos 를 쓴다.

```
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` 는 파일도 stdin 도 모른다 — `main` 이 권한을 쥔 한 줄(`read_in out 0 buf`)로 바이트를 채워 `subslice` 로 넘기면 된다. 개행 없이 끝난 마지막 조각은 `take_line` 이 주지 못하므로 `remaining` 으로 확인한다.

> **반례. `take_line` 의 `none` 을 EOF 로 단정한다**
>
> `none` 은 (a) 이미 끝이거나 (b) limit 안에 LF 가 없다는 뜻이다. 꼬리를 거두지 않으면 개행 없이 끝난 마지막 줄이 조용히 사라지고, 한 줄이 유난히 길면 그때부터 아무것도 읽지 않는 것처럼 보인다. `remaining` 으로 가른다.

> **반례. CRLF 입력의 뷰를 그대로 비교한다**
>
> `take_line` 은 CR 을 지우지 않으므로 뷰는 `"data\r"` 다. 눈으로는 같아 보이는데 `eq` 비교가 언제나 `false` 다(길이가 1 크다). 비교 전에 `strings.remove_suffix line "\r"` 로 벗긴다.

> **반례. `attach` 를 빼먹는다 · 루프 계속 여부를 `guard` 로 고른다**
>
> `spawn` 만 하고 `attach` 를 빼면 컴파일 오류가 아니라 읽기가 곧장 `none` 이다 — 입력이 있는데 0 줄이면 `attach` 부터 확인한다. `guard` 는 반드시 나가야 하므로 계속 여부를 고르면 `E-GUARD-FALLTHROUGH` 다 — `if` 로 고른다.

주의. 뷰를 오래 들고 있지 않는다 — `take_line` 이 준 슬라이스는 복사가 아니므로, 원본 버퍼를 다시 채우면 들고 있던 뷰는 다른 내용을 가리킨다. 오래 쓸 값이면 복사해 둔다. actor 핸들러를 부르는 op 도 `state` 를 선언해야 한다. 소스를 거는 핸들러가 `open` 이 아니라 `attach` 인 이유 — `files` 의 `open` 과 겹쳐 효과 해석이 엉뚱한 쪽을 집었다. 함께 쓸 모듈들과 이름이 겹치지 않게 짓는다.

outbuf — 버퍼링 출력, flush 망각은 컴파일 오류

1

소스 lib/out.low
층 L2 — 바깥 세계
권한 buf_flush · buf_write · buf_finish 에 cap io

출력을 **모아 두었다가 한 번에** 내보낸다. write_out 은 호출마다 비우므로 조각마다 부르면 조각마다 시스템 호출이다 — 이 라이터는 바이트를 호출자 버퍼에 쌓았다가 가득 찰 때만 비워 시스템 호출 수를 바이트 수에서 버퍼 수로 줄인다. 그리고 마지막에 반드시 finish 를 부르게 만들어 "모아 둔 것을 내보내지 않고 끝내는" 사고를 **컴파일 단계에서** 막는다.

```
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 .
```

소유되는 것은 버퍼가 아니라 아직 나가지 않은 바이트다. 버퍼는 호출자 것으로 두고(fmt 와 같은 규율), owned 로 지키는 것은 비워지지 않은 상태 pending 이다 — 잃으면 안 되는 바로 그것이 소유된다. finish 가 owned pending 을 받고 result 를 내므로, 그런 값을 조용히 버리면 E-OWN-INCOMPLETE 다 (19장). 어노테이션이 아니라 시그니처에서의 추론이다. 다른 언어에서는 flush 를 잊으면 출력이 통째로 사라지거나, 소멸자에서 flush 하되 오류를 버린다 — 여기는 컴파일이 되지 않는다.

라이터는 cap io 를 품지 않는다. 권한은 값이 아니라 정적 토큰이라 구조체에 담을 수 없다. 그래서 바이트가 실제로 나가는 지점마다 호출자가 권한을 댄다 — **라이터가 몰래 flush 할 수 없다**(16장).

op	모양	실패
write_all	proc (out cap io, d u64, b slice u8) → option u64 — 버퍼를 거치지 않고 한 조각을 **끝까지** 쓴다	none — 더는 못 쓴다
io_error · pending	enum write_failed · 구조체 pos u64(쌓인 바이트 수) · fd u64(1 = stdout, 2 = stderr)	—
buf_open	fn (d u64) → pending, effects none	없음 — 다만 여는 순간 갚아야 할 빚이 생긴다
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 — 가득 차면 스스로 비운다	error write_failed
buf_finish	proc (out cap io, p owned pending, buf mut slice u8) → result void io_error — 꼬리를 비우고 완결	error write_failed

표 50.1 — outbuf 의 op

[1] op 이름이 모두 buf_ 로 시작하는 까닭: 이름공간이 평평해서 open·write·finish 는 files 가 이미 쓰는 이름이다. 두 모듈이 같은 이름을 내보내면 **한 프로그램에 같이 설 수 없다** — 파일을 여는 프로그램이 버퍼링 출력을 못 쓰게 된다.

`buf_write` 와 `buf_flush` 는 소유를 소비하고 새 상태를 `ok` 로 돌려준다 — 호출자는 `set p (ok_value w)` 로 매번 이어받는다. `set` 은 사용이 아니라 재초기화라 루프 안에서도 소유가 유지된다. **짧게 쓰인 것은 실패가 아니다** — 잎(`write_out`)은 파이프가 차면 요청보다 적게 쓰고, 그것은 정상이다. `buf_flush` 는 `write_all` 로 **끝까지** 쓴다. 실패(`write_failed`)는 "더는 못 쓴다" 일 때만 나고, 그때 pending 은 소비되어 그 시점 버퍼의 바이트는 잃는다(재시도는 짓지 않았다). 버퍼가 작아도 동작은 맞고 크기는 속도에만 영향을 준다. `buf_flush` 를 직접 부를 일은 "이 줄이 당장 화면에 보여야 한다" 같은 때뿐이다.

```
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
```

순수한 포매팅(`fmt`)이 바이트를 조립하고 버퍼링 출력(`outbuf`)이 내보낸다. 조립용 버퍼 `nb` 와 라이터의 버퍼 `buf` 는 따로다.

> **반례. `finish` 생략 · 이동한 `p` 재사용**
>
> `owned pending` 을 완결하지 않고 반환하면 `E-OWN-INCOMPLETE` 다 — 중간에서 `return` 으로 빠지는 경로도 마찬가지다. `write` 뒤 `set p …` 없이 옛 `p` 를 넘기면 이동된 값의 사용으로 컴파일 오류다. 둘 다 컴파일러가 잡는다.

> **반례. `write` · `finish` 사이에 버퍼를 다른 용도로 쓴다**
>
> `pending.pos` 는 그 버퍼에 쌓인 바이트 수를 가리킨다. 조립용 버퍼와 같은 것을 쓰면 컴파일은 통과하고 나가지 않은 바이트가 덮어써져 엉뚱한 내용이 나간다. 이 모듈에서 조용히 틀리는 유일한 실수다 — 라이터 하나에 버퍼 하나를 붙박이로 쓴다.

주의. 권한 없이 비우려 하면 컴파일 오류다(`effects io` 를 선언하고 권한을 받지 않으면 `E-EFFECT-NO-CAP`). fd 는 열림이 검사되지 않는 정수다. 출력이 이미 한 버퍼에 다 있으면 라이터가 필요 없다 — `write_out` 한 번이면 된다. 벡터 쓰기(writev) · 실패한 flush 뒤의 재시도 · 중단 가능 출력은 짓지 않았다.

files — 파일 · 디렉터리 스트림, close 망각은 컴파일 오류

소스 lib/file.low
층 L2 — 바깥 세계
권한 cap file_system

파일과 디렉터리를 여닫고 읽고 쓴다. 모든 op 이 첫 인자로 cap file_system 을 받는다 — 파일시스템에 닿을 권한은 main 의 입력으로 한 번 건네지고, 넘겨받은 op 만 파일을 만질 수 있다(16장, 28장). 빌트인 잎(file_open · file_read · dir_open · file_type 등)은 커널에 닿을 뿐 정수 핸들만 안다. 정수 fd 를 그대로 들면 닫기를 잊어도 아무도 모르므로, 이 모듈이 **잊을 수 없는 값**(owned handle)으로 감싼다.

```
let n result u64 files.file_error . be files.slurp fs "data.txt" buf .
```

op	답과 그 이유
open · open_dir	result X file_error — 여는 일에는 "끝" 이 없다. 열리거나 실패다
read · read_dir	result (option u64) file_error — ok (some n) 읽었다 · ok none **끝이다(실패가 아니다)** · error e 실패다
write · seek_to · slurp	result u64 file_error — "끝" 이라는 개념이 없다
type_of · link_type_of · is_dir · is_file · is_symlink	option u64 · bool — 잎이 "없음" 과 "stat 실패" 를 한 비트로 준다. 없는 구별을 타입으로 약속하지 않는다

표 50.1 — 답의 모양은 op 마다 정직하게 다르다

file_error 의 변형도 **구별할 수 있는 것만**이다 — open_failed · read_failed · write_failed · seek_failed · close_failed, 그리고 라이브러리가 스스로 세는 buffer_too_small. 잎이 errno 를 주지 않으므로 정직한 해상도는 "어느 연산이 실패했나" 까지다.

op	하는 일
handle · dir_handle	잊을 수 없는 핸들(스칼라 fd 하나) — 파일과 디렉터리는 다른 타입
open fs path mode	연다. mode 0 읽기 · 1 쓰기(자르고 만들기) · 2 덧붙이기
read fs h buf · write fs h bytes	한 조각을 읽는다(짧은 읽기는 정상) · 쓴 바이트 수
seek_to fs h off whence	커서를 옮긴다. whence 0 절대 · 1 현재 기준 · 2 끝 기준. seek_to fs h 0 2 가 파일 크기다
close fs h	owned handle 을 소비하고 result void file_error
open_dir · read_dir fs h buf · close_dir	디렉터리 열거. read_dir 는 다음 이름을 버퍼에 넣는다(. · .. 도 나온다)
type_of · link_type_of	some 0 파일 · some 1 디렉터리 · some 2 그 밖 · 없으면 none(link_type_of 는 링크를 따라가지 않는다)
is_dir · is_file · is_symlink	편의형 — 경로가 없으면 false
slurp fs path buf	파일 전체를 버퍼로. 안 들어가면 error buffer_too_small — 자르지 않는다

표 50.2 — files 의 op — 모두 effects io, 첫 인자 fs cap file_system

close 를 잊으면 컴파일되지 않는다. `close` · `close_dir` 가 `owned` 를 받고 `result` 를 내므로 핸들을 조용히 버리면 `E-OWN-INCOMPLETE` 다. 그리고 닫기는 진짜로 실패한다 (NFS, 디스크 가득) — `result` 가 장식이 아니다. `read` · `write` 의 핸들은 비소유 파라미터라, 여러 번 읽고 마지막에 닫는 모양이 그대로 표현된다.

실패와 EOF 는 다른 답이다. 한때 `slurp` 은 읽기가 실패하면 루프를 멈출 뿐이어서 호출자가 완독과 중도 실패를 구별할 수 없었고, 줄 수를 세는 프로그램이 읽기 실패를 "0 lines" 라는 성공으로 냈다. 결함 주입기가 그것을 실물로 보였다. 지금은 실패면 `error` 를 답한다(핸들은 그래도 닫는다 — 자원 반납은 답과 별개다).

```
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
```

실패를 직접 불러 본다. `LOW_HOST_FAULT="open:err"`(모든 open 실패), `"read:err@2"`(두 번째 읽기 실패), `"read:short@1=4"`(첫 읽기를 4 바이트로 자름), `"close:err"`. VM 과 네이티브가 같은 주입기를 쓰므로 두 백엔드의 답이 같아야 한다.

> **반례. 오류 경로에서 핸들을 닫지 않는다**
>
> 열었으면 조기 반환 경로도 닫아야 한다. 빠뜨리면 `E-OWN-INCOMPLETE` 로 컴파일러가 잡는다.

> **반례. `read_dir` 결과를 거르지 않는다**
>
> `.` · `..` 이 섞여 나온다. 재귀 순회에서 거르지 않으면 같은 디렉터리를 다시 들어가 끝나지 않는다.

> **반례. `ok none` 을 실패로 처리한다 · 짧은 읽기를 끝으로 여긴다**
>
> `ok none` 은 끝이라는 **사실**이다. 반대로 부탁한 것보다 적게 읽은 `ok (some n)` 은 끝이 아니다 — 계속 읽어야 한다(`slurp` 이 그렇게 한다).

주의. 경로는 `slice u8` 바이트열이고 인코딩을 해석하지 않는다(관례상 UTF-8). `field h fd` 로 fd 를 읽어 잎을 직접 부르면 owned 규율이 지켜 주는 것이 사라진다. 버퍼 크기를 모르면 `seek_to fs h 0 2` 로 먼저 재거나 `read` 조각 루프로 바꾼다. 파일시스템 변경(디렉터리 만들기 · 지우기 · 이름 바꾸기)은 상태 없는 경로 → bool 잎이라 감싸지 않는다 — 순수 전달은 동의어일 뿐이다. 소켓은 `net` 이 맡는다. 기다리다

중간에 멈출 수 있는 입출력(wait)은 아직 없다 — 그것은 반응기(reactor)가 먼저 서야 지을 수 있다. seek 이 아니라 seek_to 인 이유 — seek 은 다른 이름과 겹쳤다.

tty — 터미널 입력

소스 lib/tty.low
층 L2 — 권한이 필요한 절반(내장) + 순수 파싱(이 모듈)
권한 내장 tty_raw · tty_read · tty_size 에 cap tty

에디터나 메뉴처럼 **키를 하나씩 받아 반응하는** 프로그램을 만들 때 쓴다. 보통 터미널은 사용자가 Enter를 눌러야 줄을 넘기고, 그 전까지 커널이 글자를 모으고 되비치고 백스페이스도 처리한다(행 버퍼링과 에코). 방향키를 누르는 즉시 커서를 옮기려면 이 편의가 방해가 되므로 **raw 모드**로 바꿔 "누른 즉시, 되비침 없이" 받는다. 그리고 방향키는 한 바이트가 아니다 — ↑ 는 ESC [A 세 바이트, Delete 는 ESC [3 ~ 네 바이트다. 그 바이트열을 키로 읽어 주는 것이 parse_key 다.

```
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 모드는 사용자의 터미널 설정을 바꾸고, 그 변경은 프로그램이 죽어도 남는다**
>
> raw 모드로 들어간 채 프로그램이 죽으면 사용자의 셸은 에코가 없는 상태로 남는다. 그런 권한을 아무 코드나 부를 수 있다면 아무 라이브러리나 남의 셸을 망가뜨릴 수 있다. 그래서 cap tty 를 든 자만 행사한다 — 권한 없이 tty_raw 를 부르면 컴파일 때 E-CAP-MISSING 이다. 동시에 키 파싱까지 권한 뒤에 두면 실제 터미널 없이는 시험할 수 없게 된다. 그래서 반을 갈랐다 — **순수한 절반이 훨씬 크다**. 권한 절반은 raw 모드 · 읽기 · 화면 크기 셋뿐이고, 키 파싱(여기)과 ANSI 조립 · 화면 diff · 표시 폭(term)은 순수하다(37장).

키코드 하나로 글자와 특수키를 구별한다. 평범한 바이트는 그 값 그대로(1 … 255, 제어 문자 포함 — Ctrl-C 는 3), 특수키는 1000 위다. 그래서 is_char 는 lt keycode 1000 한 번이다. **튜플이 없으므로** 반환은 키코드 × 16 + 길이 한 값에 담는다(길이는 1 … 8). 길이가 필요한 이유는 다음 키가 버퍼의 어디서 시작하는지 알아야 하기 때문이다. 포장 형식이 바뀌어도 코드가 깨지지 않게 key_of · len_of 로 꺼낸다.

op	계층	하는 일
key_up • key_down • key_right • key_left	순수	1001 · 1002 · 1003 · 1004
key_home • key_end • key_delete • key_pageup • key_pagedown • key_esc	순수	1005 · 1006 · 1007 · 1008 · 1009 · 1010
key_unknown	순수	1011 — 알아보지 못한 특수키
parse_key buf at	순수(effects none)	바이트열 → some (키코드 × 16 + 길이) 또는 none
key_of • len_of • is_char	순수	포장 값에서 키코드 · 길이 꺼내기 · 평범한 글자인가(**키코드**를 넣는다)
tty_raw t on**(내장)**	cap tty	raw 모드 진입(true) · 복귀(false)
tty_read t dst**(내장)**	cap tty	지금 온 바이트를 버퍼로 → option u64(some 0 = 지금은 없음, 에러가 아니다)
tty_size t**(내장)**	cap tty	화면 크기 → option u64(행 = div v 4294967296, 열 = mod v 4294967296)

표 50.1 — tty 의 op 과 내장

`parse_key` 가 알아보는 것 — ESC 아닌 한 바이트(길이 1), 뒤가 없는 `ESC`(ESC 키, 길이 1), `ESC O A–D · H · F` 와 `ESC [ A–D · H · F`(방향키 · Home · End, 길이 3), `ESC [ 1 ~ · 3 ~ · 4 ~ · 5 ~ · 6 ~`(Home · Delete · End · PageUp · PageDown, 길이 4). **불완전한 시퀀스를 지어내지 않는다** — `ESC [` 까지만 왔으면 `none` 이고 "아마 방향키겠지" 라고 추측하지 않는다. 예외는 하나 — `ESC` 하나만 오고 뒤가 없으면 ESC 키다(영원히 기다릴 수는 없다). `ESC` 뒤가 `[` 도 `O` 도 아니면 ESC 키 + 길이 1 로 답한다(Alt 조합을 ESC + 글자로 보내는 터미널에서 다음 호출이 글자를 읽는다).

```
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
```

할당처럼 실패할 수 있는 일은 **raw 로 들어가기 전에** 끝낸다 — 들어간 뒤 실패해 돌아가면 단말이 raw 로 남는다. 터미널 내장은 `cap tty` 를 **인자로** 받는다 — 권한을 op 이 가졌다는 것만으로는 안 되고 쓰는 자리에 건네야 한다. `parse_key` 는 `effects none` 이라 터미널 없이 VM · 네이티브 대조로 키 파싱 전부가 검증된다 — 반 가르기의 배당금이다.

> **반례. raw 모드를 복귀시키지 않는다**
>
> `tty_raw t true` 뒤 중간의 `guard … else return 2` 로 빠지면 복귀가 없어 셸이 망가진 채 남는다. 언어에 `defer` 가 없으므로 모든 나가는 길에서 `tty_raw t false` 를 부르거나, 실패할 수 있는 부분을 raw 진입 전에 끝낸다. 위 예제는 루프가 언제나 아래로 빠져나오게 짜여 있다.

> **반례. `none` 을 에러로 읽고 끝낸다 · 먹은 길이를 무시하고 1 씩 전진한다**
>
> `parse_key` 의 `none` 은 대개 “바이트가 더 필요하다” 다 — 읽기 경계가 시퀀스 가운데를 자를 수 있다. 여기서 끝내면 방향키를 누를 때마다 죽는 프로그램이 된다. `set off (add off 1)` 은 `ESC [ A` 를 세 개의 키로 읽는다 — 반드시 `len_of` 만큼 전진한다. 포장 값을 `1001` 과 직접 비교하면 길이가 섞여 맞지 않는다.

주의. 터미널마다 방향키 시퀀스가 달라 `ESC [ A` 와 `ESC O A` 둘 다 알아본다. 그래도 못 잡는 시퀀스는 `none` 이다 — 조용히 다른 키로 답하지 않는다. 한글을 입력하면 `parse_key` 가 **바이트 하나씩** 돌려주므로 코드포인트는 `utf8` 로 조립한다. `is_char` 의 “글자” 는 “특수키가 아니다” 라는 뜻이지 출력할 수 있다는 뜻이 아니다. raw 모드에서 Ctrl-C 는 시그널이 아니라 바이트 3 으로 온다. 화면 크기 변화(SIGWINCH)는 알려 주지 않는다 — 필요하면 매 프레임 `tty_size` 를 다시 묻는다. 마우스 보고 · bracketed paste · 터미널 질의 응답 파싱은 짓지 않았다.

net — 소켓(TCP loopback · 프로세스 안 한 쌍)

소스 lib/net.low
층 L2 — 바깥 세계
권한 cap net

소켓을 열고 · 주고받고 · 닫는다. 여는 길이 둘이다 — **TCP**(127.0.0.1 로 듣고 연결한다)와 **한 쌍**(같은 프로세스 안의 두 끝, 주소가 없다). 모든 op 이 cap net 을 받는다. 세상에 닿는 능력은 건네받는 권리이지 어디선가 꺼내 오는 것이 아니다(16장).

핸들은 자원이다. conn · listener · pair 는 닫아야 하고, **닫기는 실패할 수 있다**(result void net_error). 그래서 닫기를 잊으면 컴파일이 되지 않는다 (E-OWN-INCOMPLETE, 19장).

```
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	모양	실패
serve	(cap net, port u64) → option listener	none — 듣지 못했다
dial	(cap net, port u64) → option conn	none — 붙지 못했다
take	(cap net, l listener) → option conn	none — 받아들이지 못했다
pair_of	(cap net) → option pair	none
send_all	(cap net, fd u64, buf slice u8) → option u64	**보낸 만큼**을 답한다
recv_once	(cap net, fd u64, dst mut slice u8) → option u64	some 0 = 상대가 닫음 · none = 실패
shut • shut_listener • shut_pair	(cap net, owned X) → result void net_error	error close_failed. 쌍은 두 끝을 **각각** 닫는다
conn_fd • pair_a • pair_b	→ u64	**순수** — 권한이 필요 없다

표 50.1 — net 의 op

듣는 포트는 field l port 로 읽는다(임시 포트를 받았으면 그 번호). fd 를 읽는 것은 순수하다 — 자원은 잊지 말아야 할 것이지만, 번호를 들여다보는 데는 권한이 들지 않는다.

알아 둘 계약 둘. send_all 은 끝까지 보내되 거짓말하지 않는다 — 한 번의 송신이 전부를 보낸다는 보장이 없으므로 반복하고, 도중에 막히면 **거기까지**를 답한다. some n 을 받으면 n 이 len buf 와 같은지 확인한다. recv_once 의 0 은 실패가 아니라 상대가 닫았다는 **사실**이다.

셋을 한 번에 닫는 op 을 없앤 이유. 예전에는 핸들이 정수였고 shut3 이 fd 셋을 한 번에 닫았다. 듣기 · 연결 · 받아들인 것은 서로 다른 자원이고, 셋을 한 호출로 뭉치면 컴파일러가 **무엇을 잊었는지** 말할 수 없다. 편의가 안전을 가리면 그 편의는 값이 아니다. 지금 쌍은 두 끝을 각각 owned conn 으로 품어, 한쪽만 닫고 잊어도 컴파일되지 않는다.

시험하기 — 실패를 불러온다. `LOW_HOST_FAULT="connect:err"`(연결 거부), `"accept:err"`, `"send:err@1"`(첫 송신 실패), `"recv:short=2"`(수신을 2 바이트로 자름). VM 과 네이티브가 같은 주입기를 쓰므로 두 백엔드의 답이 같아야 한다(28장).

짓지 않은 것 — 원격 대상(loopback 만 있다), 타임아웃, 비차단 · 폴링, UDP, 주소 파싱.

clock — 시각과 마감

소스 lib/clock.low
층 L2 — 바깥 세계
권한 cap clock

"지금 몇 시인가", "얼마나 지났나", "언제까지인가" 에 답한다.

시계를 읽는 것은 io 가 아니다. 명세의 io 는 바깥과 자료를 주고받는 일(파일 · 연결 · 표준 입출력)인데, 시각을 읽는 것은 자료를 주고받지 않는다. 그래서 시계 op 은 효과를 적지 않고(effects none), 결정성을 깨는 일이라는 사실은 **권한**(cap clock)이 나른다. 예외는 sleep_ms 하나로, wait 효과다. 효과를 크게 적으면 부르는 쪽이 없는 비용을 떠안는다 — 순수한 op 이 시계를 쓰지 못하게 된다. cap random 이 같은 이유로 갈린 짝이다(16장).

```
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
```

	단조(monotonic)	벽시계(local)
op	now_ns · now_ms · since_ns · since_ms · sleep_ms · deadline_in · past · left_ms	local_packed + year_of ⋯ second_of
약속하는 것	**뒤로 가지 않는다** — 그래서 지속을 잰다	사람이 읽는 날짜 · 시각
약속하지 않는 것	절대 시각(기준점은 뜻이 없다)	단조성 — 시간대 · 서머타임 · 보정으로 뒤로 갈 수 있다

표 50.1 — 두 시계 — 이름만 비슷하고 약속이 다르다

지속을 잴 때 벽시계를 쓰지 않는다. 위 표가 있는 이유가 그것이다.

op	모양	비고
now_ns • now_ms	(cap clock) → u64	단조
since_ns • since_ms	(cap clock, start u64) → u64	지난 만큼
sleep_ms	(cap clock, ms u64) → u64	실제로 잔 시간 · effects wait
deadline_in	(cap clock, ms u64) → u64	마감 시점
past • left_ms	(cap clock, deadline u64) → bool · u64	마감 판정 · 남은 시간
local_packed	(cap clock) → u64	벽시계 여섯 값(연 · 월 · 일 · 시 · 분 · 초)을 담은 한 값
year_of ⋯ second_of	(p u64) → u64	**순수** — 푸는 짝

표 50.2 — clock 의 op

푸는 짝이 순수하므로, 시각을 **받아서** 다루는 코드는 시계 권한을 받지 않아도 된다.

확인하는 것 — 무엇을 주장할 수 있나. 시험이 주장하는 것은 단조성과 순서뿐이다. 절대 시각은 대볼 것이 없어서 주장하지 않는다 — 없는 구별을 시험으로 지어내지 않는다. **짓지 않은 것** — 시간대, 달력 산술(윤년 · 월말), 형식화, 타이머 · 알람, 단조 시계의 기준점.

trust — 신뢰 저장소에서 믿는 뿌리 찾기

소스 `lib/trust.low`
층 L2 — 호스트
권한 `cap file_system`

체인 검증의 마지막 물음은 «이 중간 인증기관을 서명한 것이 **우리가 미리 믿기로 한** 인증서인가» 다. 그 «미리 믿기로 한» 인증서들이 사는 곳이 **신뢰 저장소**이고, 리눅스에서는 PEM 인증서를 이어 붙인 묶음 파일이다. 이 모듈은 그 묶음에서 **주체 이름이 이것인** 인증서 한 장을 찾아 준다.

> **여기서 «믿는다» 는 «이 파일을 믿는다» 다**
>
> 그 파일을 누가 쓰는지는 이 모듈의 물음이 아니라, 열 권한(`cap file_system`)을 건네준 쪽의 물음이다. 그 사실을 감추지 않으려고 파일 경로를 **인자로** 받는다(안에 박아 두지 않는다).

묶음을 통째로 들지 않는다

시스템 묶음은 200 KB 가 넘는다(실측 224,449 바이트 · 인증서 150 장). 이 언어의 힙 아레나는 64 KiB 다. 그래서 **흘려 읽는다** — 창 하나를 굴리며 PEM 덩이를 하나씩 풀어 이름만 견주고 버린다. 찾은 한 장만 남긴다.

```
파일:  [ 인증서 1 ][ 인증서 2 ][ 인증서 3 ][ 인증서 4 ] …
창 1:  [ 인증서 1 ][ 인증서 2 ][ 인증서 3 의 앞]
                              └ 끝나지 않은 덩이는 이번 창에서 풀지 않는다
창 2:              [ 인증서 3 의 앞 ][ 뒤 ][ 인증서 4 ] …
                   └ 남은 꼬리를 앞으로 당기고, 그 뒤를 다음 읽기로 채운다
```

꼬리를 당기지 않으면 창 경계에 걸친 인증서 하나가 조용히 «없는 것» 이 된다 — 그리고 그 하나가 하필 우리가 찾는 뿌리일 수 있다.

op

op	하는 일
`find_anchor`	묶음 파일 `path` 에서 주체 이름이 `want`(자리 `wo`)인 인증서를 찾아 `out` 에 DER 로 담고 길이를 낸다. `effects io` — 파일을 읽는다

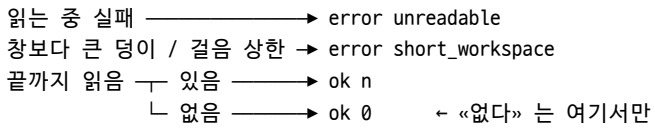
표 50.1 — `trust` 의 op

`find_anchor` 의 답은 넷이다.

답	뜻
`ok n`(n > 0)	찾았다 — `out` 의 앞 `n` 바이트가 그 인증서다
`ok 0`	끝까지 읽었고 없다 — **잰 뒤에** 하는 말은 이것뿐이다
`error short_workspace`	창(`win`, 16 KiB 넘게)이나 푸는 자리가 모자라 **끝까지 재지 못했다** — 창보다 큰 덩이를 만났거나 걸음 상한에 닿았다
`error unreadable`	묶음 파일을 열지 못했거나 읽다가 실패했다 — 경로가 틀렸는지 먼저 본다

표 50.2 — «못 찾았다» 와 «재지 못했다» 는 다르다

예전에는 아랫줄 둘이 없었다. 창이 좁아 다 못 읽어도 0, 파일을 아예 못 열어도 `ok 0` 이었다(2026-09-25에 고쳤다). 그래서 도구가 «믿는 뿌리가 없다» 고 말했는데, 그것은 **없는 것**이 아니라 **재지 못한 것**이었다.



찾은 뿌리로 마지막 마디를 잇는 일은 `verify` 의 `link_ok` 가 한다. 쓰는 모듈: `files` · `pem` · `x509` · `verify`.

찾아보기

이 책이 개념을 **정의하거나 정면으로 다루는** 자리만 모았다. 낱말이 스쳐 지나가는 자리는 싣지 않았다.

Bernstein 조건 490
Merkle DAG 513
RC11 495
array 103
bit_cast 240
bitset 153
build option 384
build tier 373
cast 150
comptime 261
ensures 163
enum 115
errors 절 125
export 249
expr 섬 94
extern 359
fn 63
guard 82
join 454
match 84
method 273
mmio 370
newtype 147
option 124
panic 86
parallel 332
pipe 285
pkg.low 383
proc 63
range 149
requires 163
result 125
same_slice 238
struct 113
task_group 318
try 127
using 234
가림 39
계약 163
계약의 등급 168
고정 창 207
고정점 455, 480
구간 분석 466
권한 185
기억 차례 336
내용 주소화 513
넓히기 48
넘침 49
단락 평가 95
단형화 263
닫개 34
두 백엔드 대조 388
뜻함수 507
레인 337
모듈 249
배타 규칙 138
부분 순서 454
분리논리 501
불변식 456
브랜드 422
빌드 모드 169
성숙도 395
세대 핸들 421
소유 221
순수함 63
순차 일관성 495
슬라이스 102
신뢰 기반 518
액터 299
영역 207
완결 224
우편함 302
원자 연산 335
인터럽트 처리기 372
잎 395
전수 검사 476
전위 표기 34
절 차례 40
좁히기 49
주변 권한 21
증명서 검산 471
지역 72
참조 136
채널 319
추상해석 456
트레이트 272
패턴 130
할당기 232
효과 174
효과 격자 484
힙 212