로우엔트

Lowent 프로그래밍 언어 표준 명세

문서 판 v0.0.32

언어 판 v1.33 · 낸 날 2026-11-13

> 이 문서는 로우엔트 언어 자체를 정의한다. 이 문서만으로 언어를 배우고 구현할 수 있어야 하며, 다른 문서를 읽어야 알 수 있는 자리를 남기지 않는다.

차례

1 범위 (Scope) 7
1.1 이 문서가 정하는 것 7
1.2 이 문서가 정하지 않는 것 7
1.3 이 언어의 목적 7
1.4 이 문서를 읽는 법 8
1.5 문법의 표기와 조항의 틀 9
2 규범 참조 (Normative references) 11
2.1 이 문서가 기대는 외부 표준 11
2.2 이 문서가 기대지 않는 것 11
3 용어와 정의 (Terms and definitions) 12
3.1 프로그램의 구성 12
3.2 값과 타입 12
3.3 계약과 검증 14
3.4 효과와 권한 14
3.5 메모리와 소유 15
3.6 처리기와 적합성 15
3.7 소스와 표기 15
3.8 계산과 문장 16
3.9 환경과 적합성 17
4 적합성 (Conformance) 20
4.1 적합한 프로그램 20
4.2 적합한 처리기 20
4.3 구현 정의와 자유 선택 21
4.4 안전한 부분집합과 신뢰 경계 22
4.5 환경 한계 22
4.6 이 문서의 판과 언어의 판 22
4.7 이 문서에 있으나 아직 처리기가 못 하는 것 23
5 환경 (Environment) 24
5.1 번역 환경과 실행 환경 24
5.2 소스 파일의 표현 24
5.3 실행 환경의 최소 요건 25
5.4 진단 26
5.5 프로그램의 시작과 끝 26
5.6 기계의 이름 27
5.7 이 단위가 무엇인지 밝히는 자리 — `package` 31
5.8 빌드마다 다르게 짓기 — `build option` 과 `config` 31
5.9 정했으되 짓지 않은 도구 — 검증된 판 · 패키지 파일 · 사설 원천 · 큰 빌드의 나눔 32
6 언어 (Language) 34
6.1 어휘 (Lexical elements) 34
6.1.1 공백과 주석 34
6.1.2 낱말 (Keywords) 35
6.1.3 이름 (Identifiers) 37
6.1.4 리터럴 (Literals) 38
6.1.5 점과 form 46
6.1.6 개행은 닫지 아니한다 (Newline is not a closer) 47

6.1.7 닫히지 않은 것 50
6.1.8 소스가 갖추어야 하는 것 50
6.2 타입 (Types) 51
6.2.1 타입의 갈래 52
6.2.2 정수 타입 52
6.2.3 참거짓 타입 55
6.2.4 부동소수점 타입 56
6.2.5 타입 사이의 변환 56
6.2.6 줄 — 배열과 슬라이스 57
6.2.7 묶음 — struct 와 enum 69
6.2.8 답을 담는 타입 — result 와 option 73
6.2.9 이름 붙인 타입 76
6.2.10 값의 안을 읽는다 — field 77
6.2.11 레인을 가진 값 — vec 78
6.2.12 비트로 이루어진 집합 — bitset 78
6.2.13 바이트 위에 얹는 눈 — 뷰 79
6.2.14 값의 범위를 좁힌 매개변수 — range 82
6.2.15 바이트 배치를 못 박는다 — layout 과 align 82
6.2.16 명시적 변환 — cast 84
6.2.17 글자를 담는 것 — 문자열은 타입이 아니다 88
6.2.18 뜻이 없는 타입 이름 88
6.2.19 값을 지닌 갈래 — 짓기와 해체 89
6.2.20 흩어진 조각을 하나로 보기 — segments 90
6.2.21 미리 끌어오기 — prefetch 91
6.2.22 타입이 맞는다는 것 91
6.2.23 칸을 적는 꼴 91
6.2.24 임의 폭 정수 — bits 92
6.2.25 정했으되 짓지 않은 타입 낱말 — c8…c64 · b8…b128 · list · 겹치는 합 · 자식 영역 92
6.3 식과 연산 (Expressions and operations) 93
6.3.1 전위 표기가 기본이다 93
6.3.2 중위 표기 — expr 섬 94
6.3.3 기본 연산과 빌트인 연산 95
6.3.4 산술의 규칙 98
6.3.5 비트 연산 (Bitwise operations) 99
6.3.6 넘쳤을 때 어떻게 할지 **이름으로 고른다** 102
6.3.7 레인을 다루는 op 103
6.3.8 크기를 묻는다 — size_of 105
6.3.9 초월 함수 105
6.3.10 참거짓은 수가 아니다 105
6.3.11 폭과 레인은 타입의 일부다 106
6.3.12 expr 섬 안의 규칙 106
6.4 선언과 계약 (Declarations and contracts) 107
6.4.1 op 의 선언 107
6.4.2 계약이란 무엇인가 109
6.4.3 계약의 여섯 절 110
6.4.4 계약을 쓰는 법 112

6.4.5 계약에 이름 주기 ... 112
6.4.6 계약이 검사를 없앤다 ... 113
6.4.7 빌드 모드가 남은 검사를 정한다 ... 113
6.4.8 이름과 범위 ... 114
6.4.9 계약의 등급 ... 116
6.4.10 지금 판정할 수 있는 위반 ... 118
6.4.11 이름이 될 수 없는 것 ... 118
6.4.12 계약이 가리키는 것 ... 119
6.4.13 시험 블록 ... 120
6.5 문장과 제어 (Statements and control flow) ... 122
6.5.1 이름을 짓는 문장 ... 122
6.5.2 조건 — `if` ... 126
6.5.3 되풀이 — `while` ... 127
6.5.4 빠져나가는 조건 — `guard` ... 132
6.5.5 돌아가기 — `return` ... 134
6.5.6 블록과 들여쓰기 ... 135
6.5.7 실패를 말하는 세 가지 길 ... 135
6.5.8 실패를 다루기 — `try` 와 소비형 ... 137
6.5.9 멈추기 — `panic` ... 137
6.5.10 나누어 도는 되풀이 — `parallel` ... 138
6.5.11 몸이 갖추어야 하는 것 ... 138
6.5.12 오류는 적은 것만 난다 ... 139
6.6 갈래를 가르기 — `match` ... 140
6.7 시험 — `test` 와 `expect` ... 143
6.8 번역 시점의 매개변수 — `comptime` ... 144
6.8.1 번역 시점에 셀 수 있어야 한다 ... 145
6.9 검사할 수 없는 자리 — `unsafe` 와 `extern` ... 150
6.9.1 기계 명령을 직접 쓰는 자리 — `asm` ... 150
6.9.2 씨와 만나는 자리 — `extern` 의 경계 ... 151
6.9.3 기계마다 다른 명령을 쓰는 자리 — `unsafe target` ... 154
6.9.4 정했으되 짓지 않은 것 — `raw` · `addr` · `adopt` · `clone` ... 155
6.9.5 정했으되 짓지 않은 특례 — 침투형 목록 · 그래프 · 한 명씩 주고받는 줄 · `layout c` ... 155
6.10 모듈 (Modules) ... 156
6.10.1 무엇을 내보내나 ... 156
6.10.2 무엇을 가져오나 ... 157
6.10.3 이름이 부딪히면 ... 158
6.10.4 선언의 차례 ... 159
6.10.5 op 안의 op — 로컬 op ... 159
6.11 타입에 붙은 op 과 트레이트 (Type-associated ops and traits) ... 160
6.11.1 타입에 op 을 붙이기 ... 160
6.11.2 트레이트 ... 160
6.11.3 붙은 op 을 부른다 — method ... 164
6.11.4 붙은 op 이 어긋나는 자리 ... 165
6.12 한 줄기로 흘리기 — `pipe` ... 165
6.12.1 융합은 최적화가 아니라 의미다 ... 169
6.12.2 필요한 만큼만 읽는다 ... 169

6.12.3 pipe 밖의 map·filter·fold — 라이브러리의 op 170
7 효과와 권한 (Effects and capabilities) 172
7.1 효과 172
7.1.1 효과 줄을 적는 법 174
7.2 권한 175
7.3 순수함이 뜻하는 것 178
7.4 다른 언어와 만나는 자리 179
7.5 원자적으로 하는 일 179
7.6 장치와 이야기하는 자리 180
7.6.1 레지스터 묶음 — mmio 181
7.7 인터럽트를 받는 자리 181
7.2.1 권한이 어긋나는 자리 182
7.2.2 들어오는 자리와 나가는 자리 184
8 메모리와 소유 (Memory and ownership) 185
8.1 값이 사는 곳 185
8.2 영역을 받는 자리 186
8.3 옮기기와 베끼기 188
8.3.1 값이 놓이는 자리 — 임시값과 목적지 188
8.4 참조 189
8.4.1 참조는 자기가 가리키는 것보다 오래 살 수 없다 189
8.5 소유와 없애기 190
8.6 메모리 안전이 어디까지 보장되는가 193
8.7 바이트를 다시 읽기 — bit_cast 와 view 193
8.7.1 bit_cast — 비트는 그대로, 읽는 법만 194
8.7.2 view — 배치를 얹는다 194
8.8 고칠 수 있음은 **적힌 대로만** 흐른다 195
8.9 이름이 같아도 타입이 다르다 195
8.10 링커가 주는 저장소 — storage reserved 195
8.11 괄호는 블록을 넘지 아니한다 196
8.12 배타 — 읽는 이 여럿 **또는** 쓰는 이 하나 196
8.13 얼로케이터 — 권한 · 정책 · 상태 200
8.14 객체마다 얼로케이터를 고르기 — op 머리의 use 절과 바인딩의 use · keep 201
9 라이브러리 (Library) 205
9.1 라이브러리도 같은 언어로 쓰인다 205
9.2 라이브러리가 지켜야 할 것 205
9.3 무엇이 언어이고 무엇이 라이브러리인가 206
9.4 얼마나 익었는가 — 성숙도의 사다리 207
9.5 무엇이 표준 라이브러리에 들어오는가 207
10 동시성 (Concurrency) 209
10.1 동시성도 효과다 209
10.2 액터 209
10.3 실행 흐름 만들기 212
10.4 흐름을 묶는 자리 — task_group 213
10.5 메시지를 나중에 배달하기 214
10.6 흐름끼리 주고받기 — 채널 216
10.7 어디까지 쓸 수 있는가 — build profile 217

10.8 기계가 감당하는 것 — `build tier` 218
10.9 나란히 도는 것이 어긋나는 자리 218
A 부록 A — 문법 요약 (Annex A: Grammar summary) 220
A.1 낱말 220
A.2 문장의 모양 220
A.3 문장 221
A.4 갈래·시험·액터 221
A.5 식 222
A.6 리터럴의 어휘 문법 222
A.7 머리 낱말과 닫개 224
A.8 없앤 낱말 225
A.9 타입 낱말 226
A.10 전체 문법 228
A.10.1 어휘 문법 228
A.10.2 번역 단위와 선언 230
A.10.3 문장 230
A.10.4 식 231
A.10.5 타입 232
B 부록 B — 진단 목록 (Annex B: Diagnostics) 234
B.1 실행 중 트랩 (E-VM-...) 234
B.2 경고 (W-...) 234
B.3 번역 잘못 (E-...) 234
C 부록 C — 용어 색인 (Annex C: Index of terms) 238
D 부록 D — 기본 연산과 빌트인 연산 (Annex D: Core ops and builtin ops) 242
D.1 이름과 뜻 242
D.1.1 기본 연산 (core op) 242
D.1.2 빌트인 연산 (builtin op) 246
D.2 받는 것과 내는 것 250
E 부록 E — 모호 목록 (Annex E: Open issues) 264
F 부록 F — 자유 선택 목록 (Annex F: Free choices) 266

1 범위 (Scope)

1.1 이 문서가 정하는 것

1 이 문서는 로우엔트 (Lowent) 프로그래밍 언어를 정의한다. 정의하는 것은 다음 넷이다.

2 가) 로우엔트로 쓴 프로그램의 표현 (representation) — 어떤 글자를 어떻게 늘어놓으면 프로그램인가.

3 나) 로우엔트 프로그램의 구문 (syntax) 과 제약 (constraints) — 어떤 늘어놓기가 올바르고 어떤 것이 올바르지 않은가.

4 다) 로우엔트 프로그램을 해석하는 의미 규칙 (semantic rules) — 올바른 프로그램이 무엇을 하는가.

5 라) 로우엔트 프로그램을 번역하고 실행하는 처리기 (processor) 가 지켜야 할 적합성 (conformance) 요건.

> **쉬운 말로**
>
> 이 문서는 "로우엔트란 무엇인가"에 답하는 유일한 문서다. 언어를 배우려는 사람도, 로우엔트 처리기를 새로 만들려는 사람도, 이 문서 하나만 읽으면 된다. 다른 문서를 찾아 읽어야 하는 자리는 남기지 않았다.

1.2 이 문서가 정하지 않는 것

1 다음은 이 문서의 범위 밖이다. 범위 밖이라는 말은 "금지한다"가 아니라 "여기서 정하지 않는다"는 뜻이다.

2 가) 로우엔트 프로그램을 만드는 방법 — 편집기, 빌드 도구, 꾸러미 관리자의 사용법.

3 나) 특정 처리기의 내부 구조 — 어떤 중간 표현을 쓰는지, 어떻게 최적화하는지.

4 다) 표준 라이브러리의 전체 목록 — §9 는 라이브러리가 지켜야 할 규칙을 정하고, 개별 함수 목록은 정하지 않는다.

5 라) 운영체제·하드웨어의 동작 — 이 문서는 그것들에 무엇을 요구하는지만 적는다(§5).

> **쉬운 말로**
>
> "정하지 않는다"와 "금지한다"를 헷갈리기 쉽다. 예를 들어 이 문서는 편집기를 정하지 않지만, 그것은 편집기를 쓰지 말라는 뜻이 아니라 **이 문서가 다룰 일이 아니라는** 뜻이다.

1.3 이 언어의 목적

1 로우엔트는 의미 엔트로피 (semantic entropy) 를 낮추도록 설계된 시스템 프로그래밍 언어다.

> **의미 엔트로피** (semantic entropy)
>
> 프로그램의 소스를 다 읽은 뒤에도 그 프로그램의 실제 거동에 대해 **남아 있는 불확실성**. 바꾸어 말하면, 그 코드가 무슨 일을 하는지 알기 위해 **소스 밖에서 가져와야 하는 정보의 양**.

2 이 언어의 모든 규칙은 그 불확실성을 줄이는 방향으로 정해졌다. 구체적으로 다음 넷을 없앤다.

3 가) **숨은 거동** — 소스에 안 적힌 일이 일어나지 않는다. 보이지 않는 메모리 할당, 보이지 않는 제어 흐름, 보이지 않는 형 변환이 없다.

4 나) **숨은 비용** — 한 줄이 얼마나 비싼지가 그 줄에 적혀 있다.

5 다) **숨은 전제** — 함수가 무엇을 요구하는지가 계약 (contract) 으로 적혀 있고, 처리기가 그것을 검사한다(§6.4).

6 라) **표기의 겹침** — 같은 뜻을 여러 방법으로 적을 수 없다. 표기가 다르면 뜻이 다르다.

7 의미 엔트로피가 낮다는 것은 소스가 **기계어와 일대일로 대응한다는 뜻이 아니다.** 효율적인 구현과 최적화를 위하여 처리기가 선택지를 가져야 하는 자리가 있다.

8 이 언어는 그런 자리를 **구분하고, 다른 부분과 격리한다.** 처리기가 고르는 자리는 모두 표시되어 있고 (**자유 선택 (free choice)**, §4.3), 자리마다 고를 수 있는 범위가 적혀 있다. 표시되지 않은 자리에서 처리기가 고르는 일은 없다.

9 그러므로 프로그램을 쓰는 사람은 **어디가 정해져 있고 어디가 열려 있는지를 알 수 있다.** 열린 자리에서는 처리기에 맡길지, 정해진 꼴로 고쳐 적을지를 **스스로 고를 수 있다.**

> **쉬운 말로**
>
> 엔트로피는 «모르는 채 남는 것» 의 양이다. 선택지가 있다는 사실만으로는 엔트로피가 늘지 않는다 — 늘리는 것은 선택지가 **어디에 있는지 모르는 것**, 그리고 **무엇까지 일어날 수 있는지 모르는 것**이다.

> **쉬운 말로**
>
> 기계에 가까운 언어라고 해서 처리기가 아무것도 고르지 못하게 묶을 수는 없다. 그렇게 묶으면 더 빠른 길을 알아도 쓰지 못한다. 반대로 고르는 자리가 언어 곳곳에 흩어져 있으면, 한 줄을 읽을 때마다 «여기서도 처리기가 무언가 고르는가» 를 물어야 한다. 그 물음이 엔트로피다.

> **쉬운 말로**
>
> 그래서 이 언어는 고르는 자리를 **한 목록에 모으고**(부록 F), 본문의 그 자리에 **표시를 붙이고**, 자리마다 **범위를 적는다.** 읽는 사람은 표시가 없는 줄에서는 적힌 그대로 일어난다고 믿을 수 있고, 표시가 있는 줄에서는 무엇이 열려 있는지 정확히 안다. 남는 불확실성이 있더라도 그 **자리와 크기를 아는 것** — 이 언어가 지향하는 것은 그것이다.

> **참고 (informative)**
>
> 이름 「로우엔트(Lowent)」 는 **low entropy**(낮은 엔트로피)의 줄임이다. 이름이 곧 판정 기준이다 — 어떤 기능을 넣을지 말지는 "그것이 의미 엔트로피를 늘리는가" 로 가른다.

1.4 이 문서를 읽는 법

1 이 문서는 규범 (normative) 과 참고 (informative) 를 가른다.

> **규범** (normative)
>
> 언어를 정하는 문장. 처리기와 프로그램이 반드시 지켜야 한다. 이 문서의 본문은 별도 표시가 없으면 전부 규범이다.

> **참고** (informative)
>
> 이해를 돕기 위한 문장. 언어를 정하지 않는다. 「참고」·「쉬운 말로」·「주의」 로 표시한 상자가 그것이다. 규범과 참고가 어긋나 보이면 **규범이 옳다**.

2 조항에는 번호가 붙어 있다. 번호는 **주소**이며, 한 번 붙인 번호의 뜻은 바뀌지 않는다. 다른 곳에서 이 문서를 인용할 때는 번호로 인용한다(예: §6.4.2).

2a 번호는 정수를 점으로 이은 모양이다(부록은 대문자로 시작한다). **다른 모양은 쓰지 아니한다** — 특히 사이에 끼워 넣으려고 3a 같은 번호를 만들지 아니한다. 조항을 사이에 넣어야 하면 **뒤의 번호를 밀어야** 하며, 그것은 이 문서가 널리 인용되기 전에만 할 수 있는 일이다.

> **참고 (informative)**
>
> "밀 수 있는 마지막 때" 라는 것이 실제로 있다. 이 문서가 아직 젊을 때는 번호를 옮기는 값이 싸지만, 다른 문서와 도구가 이 번호를 가리키기 시작하면 그때부터는 비싸다. 그래서 이 규칙은 **지금** 지켜야 값이 있다.

3 각 조항 안의 문단에도 번호가 있다(1, 2, 3 …). 정확한 인용은 조항 번호와 문단 번호를 함께 적는다(예: 6.4.2 문단 3).

4 용어는 처음 나오는 자리에서 괄호로 영어를 병기하고 뜻을 적는다. 모든 용어의 정의는 §3 에 모아 두었다.

> **주의 — 이 문서가 쓰는 조동사의 뜻**
>
> 「-하여야 한다」·「-이어야 한다」는 **요건**이다(영어 표준의 *shall*). 지키지 않으면 적합하지 않다. 「-할 수 있다」는 **허용**이다(*may*). 「-하는 것이 좋다」는 **권고**이며 지키지 않아도 적합하다(*should*). 「-하지 아니한다」는 **금지**다(*shall not*).

1.5 문법의 표기와 조항의 틀

1 이 문서는 문법을 생성 규칙 (production) 으로 적는다. 표기는 다음 하나이며 문서 전체에서 같다.

> **생성 규칙** (production)
>
> 문법의 한 낱덩이가 어떤 낱덩이들의 나열로 이루어지는지를 적은 것. `이름 ::= 몸` 의 꼴이다.

표 1 — 문법의 표기

표기	뜻
`이름 ::= 몸`	`이름` 은 `몸` 으로 이루어진다
`A B`	`A` 다음에 `B` 가 온다
`A ¦ B`	`A` 이거나 `B` 다
`[ A ]`	`A` 가 없거나 한 번 온다
`{ A }`	`A` 가 0 번 이상 온다
`( A )`	묶는다
`"글자"`	그 글자가 그대로 온다
`"a" … "z"`	`a` 부터 `z` 까지의 글자 가운데 하나
`? 글 ?`	글로 적은 뜻. 그 글이 규범이다
`이름`	다른 생성 규칙을 가리킨다

2 한 번 이상은 `A { A }` 로 적는다. 그 밖의 표기는 쓰지 아니한다.

3 생성 규칙의 이름은 영어 소문자와 붙임표로 적는다(`if-statement`). 같은 이름은 문서 전체에서 같은 규칙을 가리킨다.

3a 문법은 **두 층**이다. 어휘 문법 (lexical grammar) 은 글자의 나열이 어떤 토큰으로 갈리는지를 정한다(§6.1). 구문 문법 (syntactic grammar) 은 토큰의 나열이 어떤 문법 요소로 읽히는지를 정한다.

> **어휘 문법** (lexical grammar)
>
> 글자의 나열을 토큰의 나열로 가르는 규칙. 생성 규칙의 끝 기호가 **글자**다.

> **구문 문법** (syntactic grammar)
>
> 토큰의 나열을 문법 요소로 읽는 규칙. 생성 규칙의 끝 기호가 **토큰**이다.

3b 구문 문법의 `"if"` 는 그 철자의 토큰 하나를 뜻한다. 토큰 사이의 공백과 주석은 어휘 단계에서 버려지며 구문 문법에 나타나지 아니한다.

4 생성 규칙이 정하는 것은 **모양**뿐이다 — 어떤 낱말의 나열이 그 문법 요소로 읽히는가. 모양이 맞는 것 가운데 어떤 것이 적합한지는 생성 규칙이 정하지 아니하고, 제약과 정적 의미가 정한다.
5 생성 규칙으로 적을 수 없는 조건은 그 규칙 바로 아래에 번호 붙은 문단으로 적는다.
6 언어의 구성 요소를 정하는 조항은 다음 칸을 **이 차례로** 갖는다. 쓸 말이 없는 칸은 두지 아니한다.

표 2 — 조항의 틀

칸	무엇을 적는가
구문 (syntax)	생성 규칙. 모양만 정한다
제약 (constraints)	모양은 맞지만 적합하려면 지켜야 하는 조건. 어기면 그 프로그램은 적합하지 아니하다
정적 의미 (static semantics)	적합한 프로그램에 대해 번역할 때 정해지는 것 — 식의 타입, 이름이 가리키는 것
동적 의미 (dynamic semantics)	실행할 때 일어나는 일 — 계산하는 차례, 상태가 바뀌는 것, 멈추는 것
진단 (diagnostics)	제약을 어겼을 때 처리기가 내는 진단 식별자
예제	적합한 예와 거부되는 예. 어느 문단을 시험하는지 적는다
참고	쉬운 말로 · 주의 · 그렇게 정한 까닭. 규범이 아니다(⟦§1.4⟧)

7 제약은 요구만 적는다. 그 요구를 어겼을 때의 진단 식별자는 진단 칸이 잇는다 — 진단 식별자가 요구를 정의하지 아니한다.
8 문단 번호는 칸마다 새로 세지 아니하고 조항 안에서 이어 센다.
9 이 틀로 아직 옮기지 아니한 조항이 있다. 그런 조항은 종전의 서술을 그대로 따르며 규범의 효력은 같다.

> **참고 (informative)**
>
> 「문법 틀」 상자는 한눈에 보는 모양이다. 문법의 정의는 구문 칸의 생성 규칙이며, 둘이 어긋나 보이면 생성 규칙이 옳다.

> **쉬운 말로**
>
> 칸을 가르는 까닭은 **누구에게 하는 말인지**가 다르기 때문이다. 구문은 글을 나무로 읽는 쪽(파서)에게, 제약과 정적 의미는 그 나무가 옳은지 보는 쪽(검사기)에게, 동적 의미는 프로그램을 돌리는 쪽에게 하는 말이다. 한 문단에 셋이 섞여 있으면 읽는 사람이 그것을 갈라야 하고, 가르는 방법이 사람마다 달라진다.

2 규범 참조 (Normative references)

2.1 이 문서가 기대는 외부 표준

1 다음 문서들은 이 문서가 인용하는 범위 안에서 이 문서의 일부를 이룬다. 판이 적힌 것은 그 판이, 판이 안 적힌 것은 최신 판이 적용된다.

표 3 — 규범 참조 목록

문서	무엇을 위해 기대나
ISO/IEC 10646 (Unicode)	소스 파일의 글자 집합. 로우엔트 소스는 UTF-8 로 부호화한 유니코드 문자열이다(§5.2).
IEEE 754-2019	부동소수점 수의 표현과 연산. `f32`·`f64` 가 이것을 따른다(§6.2.4).
ISO/IEC 9899 (C)	C 언어와 주고받는 경계(§7.4). 로우엔트는 C 의 의미를 정의하지 않고, C 쪽 약속을 **인용**한다.

> **참고 (informative)**
>
> C 를 규범 참조로 두는 것은 로우엔트가 C 위에 세워졌다는 뜻이 아니다. 로우엔트 프로그램이 C 로 쓴 코드와 **같은 기계 안에서 만날 때** 지켜야 할 약속을 그쪽 표준이 이미 정해 두었기 때문이다.

2.2 이 문서가 기대지 않는 것

1 로우엔트는 다음에 기대지 아니한다. 이는 이 언어의 이식성 요건이다.

2 가) 특정 운영체제 — 운영체제가 없는 환경(프리스탠딩 (freestanding))에서도 이 문서의 언어 부분 전체가 성립한다(§5.1).

3 나) 특정 CPU 구조 — 다만 §5.3 이 정하는 **최소 요건**은 요구한다.

4 다) 실행 시간 지원 라이브러리 — 언어 자체는 어떤 런타임도 요구하지 아니한다.

5 라) 쓰레기 수집기(가비지 컬렉터 (garbage collector)) — 이 언어에는 없다(§8).

> **쉬운 말로**
>
> 이 목록이 뜻하는 바는 이렇다 — **로우엔트로 쓴 계산은 운영체제가 없는 작은 장치에서도 그대로 돈다.** 파일을 읽는 것 같은 바깥일만 환경이 허락해야 하고, 그 허락은 권한으로 건네받는다(§7.2).

> **참고 (informative)**
>
> 규범 참조가 세 개뿐인 것은 우연이 아니다. 기대는 것이 많을수록 이 언어가 돌 수 있는 자리가 줄어든다.

3 용어와 정의 (Terms and definitions)

1 이 문서에서 다음 용어는 여기 적은 뜻으로만 쓴다. 일상어의 뜻이나 다른 언어에서의 뜻과 다를 수 있으므로, 뜻이 헷갈리면 언제나 이 조항이 기준이다.

> **쉬운 말로**
>
> 처음 읽는 사람은 이 조항을 통째로 읽지 않아도 된다. 본문을 읽다가 모르는 낱말이 나오면 여기로 돌아오면 된다. 각 용어에는 그 용어를 실제로 쓰는 조항 번호를 함께 적어 두었다.

3.1 프로그램의 구성

> **프로그램** (program)
>
> 함께 번역되어 하나의 실행 단위를 이루는 로우엔트 소스 파일들의 집합. 정확한 구성은 §5.1 이 정한다.

> **모듈** (module)
>
> 이름이 붙은 소스 파일 하나. 모듈은 자기 이름을 첫 줄에 적으며, 다른 모듈에 내보낼 것을 골라 표시한다. 모듈 이름은 파일 이름과 같을 필요가 없다.

> **op** (op)
>
> 이름과 계약을 가진 실행 단위. 다른 언어의 「함수」에 해당하지만, 로우엔트에서는 **부수 효과의 유무에 따라 종류가 갈린다**(§6.4.1): 아무 효과도 내지 않는 것을 `fn`, 효과를 내는 것을 `proc` 이라 한다.

> **선언** (declaration)
>
> 이름에 뜻을 붙이는 구문. 무엇이 있는지를 말한다.

> **정의** (definition)
>
> 선언 중에서 실체까지 갖춘 것. op 의 정의는 본문을 포함한다.

3.2 값과 타입

> **값** (value)
>
> 계산의 결과로 나오는 것. 모든 값은 정확히 하나의 타입을 갖는다.

> **타입** (type)
>
> 값의 집합과, 그 값에 할 수 있는 연산을 함께 정한 것. 로우엔트의 타입은 모두 **크기와 표현이 정해져 있다** — 처리기가 마음대로 정하는 자리가 없다(§6.2).

> **폭** (width)
>
> 정수 타입이 차지하는 비트 수. `u8` 의 폭은 8 이다. 이름에 숫자가 적힌 타입의 폭은 **어느 처리기에서나 같다**. `usize`·`idiff` 만은 예외로, 그 폭은 구현이 정한다 — 16 비트 이상이고, 보통 대상 기계의 주소 폭이다(§6.2.2).

> **슬라이스** (slice)
>
> 같은 타입의 값이 연속으로 놓인 구간을 가리키는 것. 시작 위치와 **길이**를 함께 갖는다. 길이를 함께 갖는다는 점이 포인터와 다르며, 그래서 경계를 검사할 수 있다(§6.2.6).

문자 리터럴 (character literal)

작은따옴표로 감싼 한 글자(`'a'`). 값은 그 글자의 부호이며 정수다. 한 칸에 안 들어가는 것은 문자 리터럴이 아니다(§6.1.4).

문자열 리터럴 (string literal)

큰따옴표로 감싼 바이트의 줄(`"abc"`). `len` 은 **원소의 수**를 낸다 — 무엇이 원소인지는 접두사가 정한다.

이스케이프 (escape)

문자·문자열 안에서 역슬래시로 시작해 한 글자를 나타내는 표기(`\n`). 쓸 수 있는 것은 **열넷뿐**이다(§6.1.4).

접두사 (prefix)

리터럴 앞에 붙여 원소의 뜻을 정하는 낱말. `u`(UTF-16 코드 유닛) · `U`(코드포인트) **둘뿐**이다. 접두사가 없으면 바이트다.

텍스트 리터럴 (text literal)

여러 줄을 그대로 담는 문자열. `text` 다음에 태그를 적고 같은 태그가 맨 첫 칸부터 홀로 있는 줄에서 끝난다. 본문의 이스케이프를 풀지 아니한다. 다른 언어에서 heredoc · nowdoc · delimited string 이라 부르는 것에 해당한다.

부동소수점 리터럴 (floating-point literal)

소수점이나 지수를 가진 수를 소스에 직접 적은 것(`1.5` · `1e3` · `0x1p3`). 점 앞뒤에 숫자가 있어야 한다 — `.5` 와 `1.` 은 부동소수점 리터럴이 아니다(§6.1.4).

리터럴 (literal)

소스에 직접 적은 값. `42`, `true`, `"글자"` 따위.

열거 (enum)

이름 붙은 갈래들 가운데 **정확히 하나**인 값. 갈래마다 값을 실어 나를 수 있다. 갈래 이름이 곧 그 값이 무엇인지를 말하므로, 쓰이지 않는 비트열이 남는다 — 그래서 열거는 「덫 표현」을 가진다.

덫 표현 (trap representation)

그 타입의 값이 **아닌** 비트열. 그런 비트열을 그 타입의 값으로 읽는 것은 뜻이 없으며, 이 언어는 그런 읽기를 허용하지 아니한다(§8.7.1). 덫 표현이 하나도 없는 스칼라 타입을 `plain` 이라 부른다.

상표 (brand)

저장소를 **가르는 이름**. 표현이 같아도 서로 다른 저장소에서 온 값이 섞이지 않도록, `newtype` 으로 만든 타입을 그 저장소의 이름으로 삼는다. 한 상표는 저장소 하나를 가리킨다(§6.2.9).

3.3 계약과 검증

계약 (contract)

op 이 자기 입력과 출력에 대해 **스스로 적는 약속**. 여섯 절로 이루어진다 — `requires`(들어올 때 참이어야 할 것) · `ensures`(나갈 때 참인 것) · `errors`(언제 실패하나) · `effects`(무슨 효과를 내나) · `access`(무엇을 어떻게 건드리나) · `tests`(예시). 자세한 정의는 §6.4.3 에 있다.

사전 조건 (precondition)

계약의 `requires` 절이 말하는 것. op 에 들어오는 순간 참이어야 하는 조건.

사후 조건 (postcondition)

계약의 `ensures` 절이 말하는 것. op 이 정상으로 끝날 때 참인 조건.

검사 (check)

실행 중에 조건이 참인지 확인하는 일. 처리기가 그 조건이 **언제나 참임을 증명하면** 그 검사는 없앤다. 없애지 못하면 남는다(§6.4.6).

트랩 (trap)

조건이 깨졌을 때 프로그램을 **즉시 멈추는 것**. 로우엔트는 잘못된 상태로 계산을 계속하지 아니한다 — 틀린 답을 돌려주느니 멈춘다.

미정의 동작 (undefined behaviour)

소스가 정하지 않은 거동. **안전한 부분집합**(§4.4)에는 없다 — 그 안에서 이 문서가 정의하지 않은 상황은 전부 진단(§5.4)이나 트랩으로 끝난다. 이것이 C 와 가장 크게 갈리는 자리다.

안전한 부분집합 (safe subset)

`unsafe` 와 `extern` 을 쓰지 않는 프로그램의 집합. 이 집합 안에서는 미정의 동작이 없다.

빌드 모드 (build mode)

증명하지 못한 계약 검사를 실행 중에 둘지 없앨지 정하는 설정. 소스에 `build <모드>` . 로 적는다(§6.4.7).

신뢰 경계 (trust boundary)

언어의 보장이 끝나고 사람의 약속이 시작하는 자리. `unsafe` 와 `extern` 이 그 자리를 소스에 표시한다(§6.9).

3.4 효과와 권한

효과 (effect)

op 이 바깥 세상에 주는 영향. 파일을 읽는 것, 시계를 보는 것, 메모리를 할당하는 것, 다른 실행 흐름을 만드는 것 따위가 각각 효과다. op 은 자기가 내는 효과를 계약에 적어야 하며, 적은 것보다 많은 효과를 내면 번역이 거부된다(§7.1).

순수 (pure)

아무 효과도 내지 않음. 순수한 op 은 같은 입력에 언제나 같은 결과를 낸다.

권한 (capability)

효과를 낼 수 있는 자격을 나타내는 **값**. 로우엔트에는 아무나 쓸 수 있는 전역 권한이 없다 — 파일을 읽으려면 파일 권한을 인자로 **건네받아야** 한다(§7.2).

3.5 메모리와 소유

영역 (region)

값이 사는 메모리 구획. 각 값은 어느 영역에 사는지가 타입과 함께 정해진다(§8.1).

참조 (reference)

다른 곳에 있는 값을 가리키는 것. 읽기 전용 참조와 쓰기 참조가 구별되며, 한 값에 대해 쓰기 참조는 한 번에 하나만 있을 수 있다(§8.4).

소유 (ownership)

어떤 값을 없앨 책임이 누구에게 있는가. 로우엔트는 소유자를 정적으로 추적하므로 두 번 없애거나 없애기를 잊는 일이 번역 단계에서 거부된다(§8.5).

탈출 (escape)

지역 값을 가리키는 참조가 그 값보다 오래 사는 것. 로우엔트는 이것을 번역 단계에서 거부한다(§8.4.1).

3.6 처리기와 적합성

처리기 (processor)

로우엔트 프로그램을 번역하거나 실행하는 것의 총칭. 컴파일러도, 해석기도, 둘을 겸한 것도 처리기다. 이 문서는 그 내부를 정하지 않고 **관측 가능한 거동**만 정한다.

번역 (translation)

소스를 실행할 수 있는 형태로 바꾸는 일. 번역 중에 발견되는 잘못은 진단으로 알린다.

진단 (diagnostic)

처리기가 잘못을 알리는 메시지. 안정된 식별자(예: `E-TYPE-INSTANCE`)를 가지며, 그 식별자는 판이 올라가도 뜻이 바뀌지 아니한다(§5.4).

적합한 프로그램 (conforming program)

이 문서의 구문과 제약을 전부 지키는 프로그램(§4.1).

적합한 처리기 (conforming processor)

적합한 프로그램을 이 문서가 정한 대로 번역·실행하고, 적합하지 않은 프로그램에 대해 진단을 내는 처리기(§4.2).

3.7 소스와 표기

로우엔트 (Lowent)

이 문서가 정의하는 프로그래밍 언어. 이름은 **low entropy**(낮은 엔트로피)의 줄임이다.

토큰 (token)

소스를 나눈 가장 작은 조각. 낱말·이름·리터럴·구두점이 토큰이다.

낱말 (keyword)

언어가 뜻을 정해 둔 이름. 다른 뜻으로 쓸 수 없다. 목록은 부록 A 에 있다.

이름 (identifier)

저자가 짓는 이름. ASCII 영문자·숫자·밑줄로 이루어진다 — **점은 들어가지 않는다.** 점은 값의 필드를 고르거나 모듈을 밝힐 때 두 이름을 잇는 표시다(§6.1.3).

주석 (comment)

`rem` 부터 줄 끝까지. 처리기가 없는 것으로 다룬다.

폼 (form)

전위 표기의 한 덩어리. 닫개가 그것이 끝났음을 알린다.

닫개 (closer)

폼이 끝났음을 알리는 표시. 마침표 `.` **하나**다. 괄호의 `)` 는 안에 열린 폼을 함께 닫고, `end` 는 자기 `do` 만 닫는다(§6.1.6).

개행 (newline)

줄이 바뀌는 자리. 사이띄개와 똑같이 다루며 폼을 닫지 **아니한다**(§6.1.6).

전위 표기 (prefix notation)

연산의 이름이 먼저 오고 인자가 뒤에 오는 표기. 우선순위 규칙이 없다.

중위 표기 (infix notation)

연산 기호가 인자 사이에 오는 표기. `expr` 섬 안에서만 쓸 수 있다.

섬 (island)

중위 표기가 켜지는 자리. `expr` 로 시작한다.

BOM (byte order mark)

일부 편집기가 파일 첫머리에 넣는 표시(U+FEFF). 로우엔트 소스에는 올 수 없다.

3.8 계산과 문장

식 (expression)

값을 만들어 내는 것.

문장 (statement)

실행되는 것. 식은 값을 만들고 문장은 일을 한다.

기본 연산 (core op)

언어가 뜻을 정해 둔 연산 가운데, 어디서나 제 이름만으로 부르는 것. 저자의 이름과 한 이름공간을 쓰므로 그 이름을 가릴 수 없다.

빌트인 연산 (builtin op)

언어가 뜻을 정해 둔 연산 가운데, `call_builtin` 뒤에서만 부르는 것. 제 이름공간을 따로 가지므로 저자의 이름과 부딪치지 않는다. 시스템 라이브러리와 처리기를 짓는 데 쓰는 것이고, 보통의 프로그램은 그것을 감싼 라이브러리를 쓴다.

단락 평가 (short-circuit)

앞쪽만으로 결과가 정해지면 뒤쪽을 계산하지 않는 것. `and` 와 `or` 가 그렇다.

넓히기 (widening)

값을 잃지 않는 타입 변환. 자동으로 일어날 수 있다.

좁히기 (narrowing)

값을 잃을 수 있는 타입 변환. 소스에 적어야 하며, 값이 안 들어가면 트랩한다.

트레이트 (trait)

어떤 타입이 갖춰야 할 op 의 목록. 타입이 그것을 갖췄다고 적으면 처리기가 검사한다. 타입 사이에 위아래를 만들지 않는다 — 이 언어에는 상속이 없다.

액터 (actor)

자기 상태를 가진 실행 단위. 상태는 액터 안에만 있다.

3.9 환경과 적합성

번역 환경 (translation environment)

소스를 실행할 수 있는 형태로 바꾸는 자리.

실행 환경 (execution environment)

그 결과가 실제로 도는 자리.

시작점 (entry point)

실행되는 프로그램이 시작하는 op. 이름은 `main` 이다.

종료 상태 (exit status)

시작점이 돌려주는 값. 0 은 성공을 뜻한다.

라이브러리 모듈 (library module)

시작점이 없는 번역 단위. 다른 프로그램이 가져다 쓴다.

적합성 (conformance)

이 문서가 정한 것을 지켰는가. 프로그램에 대해서도, 처리기에 대해서도 쓴다.

구문 (syntax)

어떤 글자의 늘어놓기가 프로그램인가에 대한 규칙.

제약 (constraints)

구문에 맞더라도 지켜야 하는 규칙. 어기면 진단이 나온다.

의미 규칙 (semantic rules)

올바른 프로그램이 무엇을 하는가에 대한 규칙.

표현 (representation)

값이 기계 안에서 어떤 비트로 놓이는가.

옥텟 (octet)

8 비트. 이 언어에서 바이트는 언제나 옥텟이다.

투스 컴플리먼트 (two's complement)

부호 있는 정수를 나타내는 방식. 이 언어가 요구하는 유일한 방식이다.

자리 (place)

값을 **쓸 수 있는** 곳. 지역 변수, struct 의 필드, 줄의 한 칸이 자리다. set 의 왼쪽에 올 수 있는 것이 곧 자리다 — 읽기만 되는 것은 자리가 아니다.

한정 (qualification)

이름 앞에 그 이름이 사는 곳을 붙여 **어느 이름인지 하나로 정하는 것**. 이 언어에서 한정은 이름에 **붙은** 점으로 적으며, 뜻은 셋뿐이다: 모듈, 변형, 그리고 타입에 딸린 선언. 값의 안을 읽는 것은 한정이 아니다.

목적지 (destination)

값이 만들어져 놓일 자리. §8.3.1 의 세 조건이 맞으면 값은 중간 임시값 없이 여기에 **곧바로** 만들어진다.

임시값 (temporary)

목적지가 없거나 조건이 안 맞을 때 만들어지는, 이름 없는 자리의 값. 그것을 감싸는 가장 작은 범위에 살고, 범위가 끝날 때 **만든 차례의 거꾸로** 없어진다.

가비지 컬렉터 (garbage collector)

쓰지 않는 메모리를 실행 중에 찾아 되돌려 주는 장치. **이 언어에는 없다.**

스테이지 (stage)

§6.12 의 `pipe` 안에서 원소 하나를 받아 다음으로 넘기는 자리. 어휘가 닫혀 있어(일곱) 그 밖의 연산은 스테이지가 될 수 없다.

종결자 (terminal)

`pipe` 를 끝내며 결과를 내는 자리. 한 `pipe` 는 종결자를 **정확히 하나** 갖는다.

문맥 (context)

`pipe` 의 스테이지·종결자에 `with` 로 붙여, 훑기 전에 한 번 계산해 op 의 마지막 인자로 건네는 값 (§6.12 (7)). 참조 없는 값만 문맥이 될 수 있다.

로컬 op (local op)

op 본문 바로 아래에 선언해 그 op 안에서만 보이는 op(§6.10.5). 바깥의 이름을 보지 않으므로 클로저가 아니다.

최적화 (optimisation)

뜻을 바꾸지 아니하면서 프로그램을 빠르게 하거나 작게 하는 처리기의 재량. **재량이므로 프로그램이 그것에 기대어서는 아니 된다** — 기대야 하는 성질은 최적화가 아니라 의미로 적힌다(§6.12 의 융합이 그 예다).

4 적합성 (Conformance)

4.1 적합한 프로그램

1 적합한 프로그램 (conforming program) 이란 이 문서의 구문(§6) 과 제약(§6) 을 전부 지키는 프로그램을 말한다.

2 적합한 프로그램은 이 문서가 정하지 않은 거동에 기대지 아니한다.

2a 진단에는 두 갈래가 있다. **번역을 막는 것**(이름이 `E-` 로 시작한다)과 **막지 않는 것**(`W-·NOTE-`)이다. 적합함을 정하는 것은 앞의 것뿐이며, 뒤의 것이 있어도 프로그램은 적합할 수 있다.

2b 곧 적합한 프로그램이란 **번역을 막는 진단이 하나도 없는** 프로그램이다.

> **예제 (example) — 적합한 프로그램**
>
> **소스 코드 (source)**
>
> ```
> module ex_conform .
>
> rem 이 프로그램은 이 문서가 정한 것을 전부 지킨다.
> export fn clamp input n u32 . output u8 .
>   requires le n. 255 . .
>   ensures le ret. 255 . .
> do
>   return narrow u8 n. . .
> end .
>
> proc main output u8 . effects none .
> do
>   return 0 .
> end .
> ```

> **쉬운 말로**
>
> 위 프로그램을 `clamp 200` 으로 부르면 200 이 나온다. `clamp 300` 으로 부르면 — 계약이 *"255 이하"* 라 했으므로 — **그 자리에서 멈춘다**(`E-VM-CONTRACT`). 틀린 답을 돌려주지 않는다는 것이 이 언어의 약속이고, 그 약속은 실행에서 지켜진다.

> **주의 — 적합하다는 것과 옳다는 것은 다르다**
>
> 적합한 프로그램도 틀린 답을 낼 수 있다. 적합성은 **언어의 규칙을 지켰는가**이지 **의도한 일을 하는가**가 아니다. 뒤엣것은 계약(§6.4.3)과 시험이 다룬다.

4.2 적합한 처리기

1 적합한 처리기 (conforming processor) 는 다음을 모두 만족하여야 한다.

2 가) 적합한 프로그램을 이 문서가 정한 의미대로 번역하고 실행하여야 한다.

3 나) 적합하지 않은 프로그램에 대하여 **적어도 하나의 진단**을 내야 한다. 진단을 낸 뒤에 번역을 계속할지는 처리기가 정한다.

4 다) 이 문서가 정한 진단 식별자를 쓰는 경우, 그 뜻을 바꾸어 쓰지 아니한다.

5 라) 이 문서가 정하지 않은 확장을 제공하는 경우, 그 확장을 쓰지 않은 적합한 프로그램의 의미를 바꾸지 아니한다.

> **참고 (informative)**
>
> 나) 는 *"모든 잘못을 찾아내야 한다"* 는 뜻이 아니다. 적합하지 않은 프로그램 하나에 대해 진단이 **하나라도** 나오면 된다. 다만 로우엔트는 검사를 미루지 않는 것을 원칙으로 하므로, 실제 처리기는 대개 잘못을 전부 모아서 낸다.

4.3 구현 정의와 자유 선택

1 이 문서가 처리기에 재량을 남기는 자리는 **두 갈래**이고, 그 자리는 **명시적으로 표시**한다.

> **구현 정의** (implementation-defined)
>
> 처리기가 정하되 **문서로 적어야** 하는 것. 예: 최대 재귀 깊이. 프로그램은 그 값을 물어볼 수 있고, 처리기마다 다를 수 있다.

> **자유 선택** (free choice)
>
> 최적화나 처리기의 사정 때문에 **고르는 것을 자유롭게 둘 필요가 있는 자리**. 이 문서가 그 자리에서 허용하는 둘 이상의 동작이나 결과 가운데 하나를 처리기가 고른다. 처리기는 어느 것을 골랐는지 **문서로 적지 않아도 된다**.

1a **자유 선택 (free choice)** 의 자리마다 이 문서는 **고를 수 있는 범위**를 적는다. 처리기는 그 범위 안에서만 고른다.

1b 처리기는 같은 프로그램의 같은 자리에서도 번역할 때마다, 실행할 때마다 다르게 고를 수 있다.

1c **자유 선택 (free choice)** 은 **정해지지 않은 것이 아니다.** 범위 안의 어느 것이 골라지든 그것은 이 문서가 정한 동작이다.

1d **자유 선택 (free choice)** 의 자리는 본문에 «자유 선택» 상자로 표시한다. 부록 F 가 그 전부를 모으며, 부록 F 에 없는 자리는 **자유 선택 (free choice)** 이 아니다. 자리를 더하거나 범위를 넓히는 것은 이 문서의 개정이다.

1e 적합한 프로그램은 **자유 선택 (free choice)** 에서 처리기가 **어느 것을 고르는가**에 기대지 아니한다(§4.1 (2)).

1f 이 문서가 **아직 정하지 않은** 자리가 있다. 그런 자리는 재량이 아니다 — 처리기에 주어진 선택이 아니라 이 문서가 앞으로 정해야 하는 것이다. 본문에 «모호» 상자로 표시하고 부록 E 가 모은다.

> **모호** (open issue)
>
> 이 문서가 **아직 정하지 않았고 나중에 정해야 하는** 자리. 가능한 풀이를 나란히 적어 둔다. 정해지면 규범 문단이 되거나, 열어 두기로 정해지면 자유 선택이 된다.

1g 모호와 **자유 선택 (free choice)** 은 다르다. 모호는 **미결**이고, **자유 선택 (free choice)** 은 **결정**이다 — 열어 두기로 정한 것이다. 적합한 프로그램은 모호의 자리에도 기대지 아니한다(§4.1 (2)).

2 이 두 갈래 밖의 재량은 없다. **안전한 부분집합**(§4.4)에서는 미정의 동작이 존재하지 아니한다 — 그 안의 어떤 프로그램도 「무슨 일이 일어날지 모른다」는 상태로 실행되지 아니한다.

> **쉬운 말로**
>
> 두 갈래를 쉬운 말로 적으면 이렇다. **구현 정의**는 «처리기마다 다를 수 있지만, 무엇으로 정했는지 처리기가 적어 둔다» 이다. **자유 선택**은 «이 가운데 아무거나 골라도 되고, 무엇을 골랐는지 적지 않아도 된다» 이다.

> **쉬운 말로**
>
> 자유 선택을 두는 까닭은 처리기가 빠른 길을 고를 수 있게 하려는 것이다. 예를 들어 `add f x. . g y. . .` 에서 `f` 와 `g` 가운데 무엇을 먼저 계산해도 답은 같다. 차례를 못 박아 두면 처리기는 더 빠른 차례를 알아도 쓰지 못한다. 그래서 그 차례를 열어 둔다.

> **쉬운 말로**
>
> 열어 둔다고 해서 무슨 일이든 일어날 수 있는 것은 아니다. 자리마다 **고를 수 있는 범위가 적혀 있고**, 그 밖의 일은 일어나지 아니한다. 그리고 그런 자리는 부록 F 의 목록이 전부다 — 목록에 없는 자리에서 처리기가 제 마음대로 고르는 일은 없다. 범위는 좁게, 자리 하나하나를 따져서 정한다.

> **쉬운 말로**
>
> 프로그램을 쓰는 사람이 기억할 것은 하나다: **어느 것이 골라지는지에 기대지 않는다.** 차례가 중요하면 차례가 정해진 꼴로 적는다(§6.3.1 의 상자가 그 방법을 말한다).

4.4 안전한 부분집합과 신뢰 경계

1 안전한 부분집합 (safe subset) 은 `unsafe` 와 `extern` 을 쓰지 않는 프로그램의 집합이다. 앞 조항의 보장은 이 집합에 대한 것이다.

2 `unsafe` 와 `extern` 은 신뢰 경계 (trust boundary) 를 표시한다(§6.9). 그 자리에서 언어의 보장이 끝나고 **사람이 적은 약속**이 시작한다.

3 경계 밖의 약속이 참이면 프로그램 전체의 거동이 이 문서가 정한 대로다. 약속이 거짓이면 이 문서는 아무것도 보장하지 아니한다 — 그때 무슨 일이 일어나는지는 정해지지 않는다.

4 그러므로 경계는 **좁아야 한다**. 넓힐수록 이 문서의 보장이 닿지 않는 넓이가 는다.

> **쉬운 말로**
>
> 이 조항이 있는 까닭은 정직함 때문이다. 다른 언어로 쓴 함수를 부르면서 "우리 언어에는 미정의 동작이 없다"고 말할 수는 없다 — 그 함수가 무엇을 할지 이 문서는 모른다. 말할 수 있는 것은 **어디까지가 언어의 보장이고 어디부터가 사람의 약속인가**이며, `unsafe` 와 `extern` 은 그 자리를 **소스에 남긴다**. 나중에 찾을 수 있다는 것이 보장만큼 중요하다.

4.5 환경 한계

1 처리기는 자원의 한계를 가질 수 있다. 그러나 한계에 부딪혔을 때 **조용히 잘라내서는 아니 된다** — 진단을 내거나 트랩하여야 한다.

2 처리기는 자기가 지원하는 한계를 문서로 적어야 한다(구현 정의).

3 한계에 부딪혔을 때 나는 진단은 계열로 묶여 있다 — 처리기 안쪽의 표가 넘치면 `E-IR-…`·`E-TYPE-LIMIT`·`E-PARSE-LIMIT` 이, 실행 중의 자원이 다하면 `E-VM-…` 이 난다. 이 문서는 그 하나하나를 적지 아니하며, **적어야 할 것은 (1) 과 (2)** — 조용히 자르지 않고, 한계를 문서로 적는 것이다.

> **쉬운 말로**
>
> 진단의 이름을 규범에 하나씩 박아 넣지 않는 까닭은, 그것이 **처리기의 속사정**이기 때문이다. 표 하나를 둘로 나누면 이름도 바뀌는데, 그때마다 규범을 고쳐야 한다면 그 규범은 처리기를 따라다니는 그림자일 뿐이다. 규범이 정할 것은 **무엇을 약속하는가** 이고, 그 약속은 「넘치면 말한다」와 「얼마까지인지 적는다」 둘이다.

> **주의 —**
>
> 한계를 적는 일과 한계를 **지키는** 일은 다르다. 적기만 하고 넘칠 때 조용히 자르면 (1) 을 어긴 것이고, 지키기만 하고 적지 않으면 쓰는 사람은 **부딪히고 나서야** 그것이 있었음을 안다. 둘 다 있어야 한다.

> **참고 (informative)**
>
> "조용히 잘라내지 않는다"가 이 조항의 요점이다. 한계 자체는 어느 처리기에나 있다. 문제가 되는 것은 한계를 넘었는데 **아무 말 없이 계속 도는 것**이며, 그때 프로그램의 결과는 소스가 말한 것과 달라진다 — 그것이 곧 의미 엔트로피다(§1.3).

4.6 이 문서의 판과 언어의 판

1 이 문서는 언어와 **별개의 판 번호**를 갖는다. 겉장에 적힌 「문서 판」이 이 문서의 판이고, 「언어 판」이 이 문서가 기술하는 언어의 판이다.

2 문서 판의 뒷자리가 오르는 것은 **서술이 늘거나 고쳐진 것**이며, 언어가 바뀐 것이 아니다. 언어가 바뀌면 언어 판이 오르고, 이 문서의 가운데 자리 이상이 함께 오른다.

4.7 이 문서에 있으나 아직 처리기가 못 하는 것

1 이 문서가 정한 것 가운데 처리기가 **아직 하지 못하는** 것이 있을 수 있다. 그때 처리기는 그것을 **없는 것처럼 다루지 아니한다.**

2 그런 자리에서 처리기는 "이 낱말은 명세에 있고, 못 하는 것은 처리기 쪽이다"라고 말하여야 한다. 곧 그것은 프로그램의 잘못이 아니다.

3 이 구별이 규범인 까닭은 **사람이 무엇을 할지가 달라지기** 때문이다. "없는 이름"이라 하면 사람은 없는 오타를 찾으러 가고, "아직 못 한다"하면 다른 길을 찾거나 기다린다. 진단 하나가 사람의 반나절을 정한다.

4 처리기는 그런 자리를 **셀 수 있게** 하여야 한다 — 모든 정적 검사를 지났으나 실행으로 내려가지 못하는 op 이 무엇인지 물으면 답하여야 한다.

> **쉬운 말로**
>
> 이것은 «미완성을 봐 준다» 는 조항이 아니다. 그 반대다 — 미완성을 **드러내라**는 조항이다. 못 하는 것을 조용히 못 하는 처리기와, 못 한다고 말하는 처리기 사이에는 신뢰의 차이가 있다.

5 환경 (Environment)

1 이 조항은 로우엔트 프로그램이 **번역되고 실행되는 자리**를 정한다. 언어의 문법은 §6 이 정하고, 여기서는 그 문법이 놓이는 바탕을 정한다.

5.1 번역 환경과 실행 환경

1 번역 환경 (translation environment) 은 로우엔트 소스를 실행할 수 있는 형태로 바꾸는 자리다. 실행 환경 (execution environment) 은 그 결과가 실제로 도는 자리다. 둘은 같은 기계일 수도, 다른 기계일 수도 있다.

> **번역 단위** (translation unit)
>
> 함께 번역되는 소스 파일 하나. 로우엔트에서 번역 단위는 곧 모듈 하나이며, 모듈은 자기 이름을 첫 줄에 적는다.

> **프로그램** (program)
>
> 함께 묶여 하나의 실행 단위를 이루는 번역 단위들의 집합. 프로그램에는 시작점이 정확히 하나 있어야 한다.

2 실행 환경은 두 갈래로 나뉜다.

> **호스티드** (hosted)
>
> 운영체제가 있는 환경. 파일·시계·표준 입출력 따위를 권한으로 받아 쓸 수 있다.

> **프리스탠딩** (freestanding)
>
> 운영체제가 없는 환경. 언어 자체는 그대로 성립하며, 다만 호스트가 주는 권한 (§7.2)을 받을 수 없다.

3 이 문서의 §6 부터 §8 까지는 **두 환경 모두에서** 성립한다. 환경에 따라 달라지는 것은 어떤 권한을 받을 수 있는가뿐이다.

> **쉬운 말로**
>
> 쉽게 말하면 — 로우엔트로 쓴 계산은 운영체제가 없는 작은 장치에서도 **문법과 의미가 똑같다**. 달라지는 것은 「파일을 읽을 수 있는가」 같은, 바깥세상과 닿는 부분뿐이다.

5.2 소스 파일의 표현

1 로우엔트 소스 파일은 UTF-8 로 부호화한 유니코드 문자열이어야 한다. 다른 부호화는 적합하지 아니하며, 처리기는 진단을 내야 한다.

2 줄 끝은 줄바꿈 문자(U+000A)로 나타낸다. 처리기는 캐리지 리턴(U+000D)이 그 앞에 붙은 경우를 받아들일 수 있으며, 그때 두 글자를 한 줄 끝으로 다룬다.

3 바이트 순서 표시(BOM (byte order mark), U+FEFF)는 소스 파일의 어느 자리에도 올 수 없다. 첫머리에 있어도 **어휘 오류**이며, 처리기는 진단을 내야 한다.

4 이름 (identifier) 에는 ASCII 문자만 쓸 수 있다(§6.1.3). 주석·문자열· 텍스트 리터럴의 **내용**에는 유니코드 문자를 자유롭게 쓸 수 있다.

5 소스가 읽히는 부호화(처리기가 읽는 바이트)와 리터럴이 갖는 값의 부호화(프로그램이 다루는 바이트)는 **별개**다. 소스는 언제나 UTF-8 이며, 리터럴이 UTF-8 이 아닌 바이트를 값으로 가질 수 있다.

> **주의 — 이름이 ASCII 인 것은 제약이 아니라 결정이다**
>
> 유니코드 이름을 허용하면 **같아 보이는데 다른 이름**이 생긴다(그리스 문자 A 와 라틴 A, 정규형이 다른 두 한글 표기 따위). 그러면 소스를 읽은 사람이 틀린 결론에 이르는데, 그것이 이 언어가 없애려는 불확실성이다(§1.3). 그래서 이름은 ASCII 로 못박고, 사람이 읽을 글은 주석·문자열에 유니코드로 적는다.

> **쉬운 말로**
>
> 한글로 함수 이름을 지을 수는 없지만, 주석과 문자열은 한글로 마음껏 쓸 수 있다. 실제로 이 저장소의 예제들이 그렇게 쓰여 있다.

5.3 실행 환경의 최소 요건

1 적합한 실행 환경은 다음을 만족하여야 한다.

2 가) 바이트(옥텟 (octet))는 정확히 8 비트여야 한다.

3 나) 정수는 2 의 보수(투스 컴플리먼트 (two's complement)) 로 표현되어야 한다.

4 다) 주소의 폭은 16 비트 이상이어야 한다.

4a **포인터는 주소와 같은 것이 아니다.** 포인터 값은 주소에 더해 경계나 권한을 실을 수 있고, 그래서 그 크기는 주소의 폭보다 **클 수 있다**. 포인터 값의 크기와 정렬, 그리고 그것이 능력 (capability) 인지 (정수에서 만들 수 없고 다른 유효한 포인터에서만 파생되는 값인지)는 실행 환경이 정한다(구현 정의). 프로그램은 그 값들을 `machine.addr_bits` · `machine.ptr_bytes` · `machine.ptr_align` · `machine.ptr_caps` 로 번역 시점에 묻는다(`machine.ptr_width` 는 `machine.addr_bits` 와 같다).

> **능력** (capability)
>
> 주소와 함께 **닿아도 되는 범위와 할 수 있는 일**(읽기 · 쓰기 · 실행)을 싣는 포인터 값. 하드웨어가 그 값이 유효한지를 따로 기억하므로 정수에서 만들거나 고쳐 쓸 수 없고, 다른 유효한 능력을 **좁혀서만** 얻는다. 범위 밖에 닿으면 실행이 멈춘다.

4b 이 언어의 값은 포인터를 **정수로 내보이지 아니한다.** 가리키는 것은 슬라이스 · 참조 · 권한이고, 어떤 정수 타입도 포인터를 담았다가 되살리지 못한다. 그러므로 포인터가 능력인 기계에서도 프로그램의 뜻은 바뀌지 아니한다.

> **참고 (informative)**
>
> 포인터가 주소보다 큰 기계가 실제로 있다 — 능력 포인터 기계(주소 64 비트 · 포인터 128 비트, 또는 주소 32 비트 · 포인터 64 비트)는 포인터에 경계와 권한을 싣고, 정수를 거친 포인터는 쓸 수 없게 만든다. 주소 공간마다 포인터 크기가 다른 기계 (프로그램 메모리와 데이터 메모리가 나뉜 마이크로컨트롤러, 가속기의 공간들)도 있다. (4a) · (4b) 는 그런 기계를 맞는 날 언어의 약속이 먼저 깨지지 않도록 적어 둔 것이다.

4c **곱셈기의 폭**은 두 정수의 곱을 위 절반까지 명령 하나로 내는 가장 넓은 폭(8 · 16 · 32 · 64 비트)이다. 그것은 실행 환경이 정하고(구현 정의), 프로그램은 `machine.mul_bits` 로 번역 시점에 묻는다. 번역기는 번역 하나에서 그 값을 바꾸는 수단을 줄 수 있다. 그 값은 주소의 폭과 다를 수 있다. 처리기는 큰 수의 셈에 쓰는 **조각의 폭**도 정해 `machine.limb_bits` 로 답한다(16 · 32 · 64 비트, 구현 정의) — 기계에서 재어 정한 값이 있으면 그 값이고, 없으면 곱셈기의 폭이다(곱셈기의 폭이 16 비트보다 좁으면 16 비트). 번역기는 번역 하나에서 조각의 폭을 바꾸는 수단을 줄 수 있다. 곱셈기의 폭은 기계의 사실이라 바뀌지 아니한다.

> **참고 (informative)**
>
> 큰 수의 셈은 수를 조각으로 나눠 조각끼리 곱한다. 어떤 폭의 조각이 빠른지는 주소의 폭이 아니라 곱셈기가 정한다 — 주소가 32 비트여도 32×32 비트 곱의 위 절반을 내지 못하는 기계가 있다(Cortex-M0). 그런데 잰 결과가 곱셈기 폭과 다를 수 있어 (32 비트 x86 빌드에서는 64 비트 조각이 더 빨랐다) 고른 폭은 따

> 로 둔다. 표준 라이브러리의 큰 수 모듈 `bigint` 는 `machine.limb_bits` 로 조각 폭을 고르고, 어느 폭에서든 같은 답을 낸다(RFC-0152).

5 라) 부동소수점 수를 쓰는 프로그램에 대하여, 실행 환경은 IEEE 754 의 이진 32 비트 또는 64 비트 형식을 제공하여야 한다. 부동소수점 수를 쓰지 않는 프로그램은 이 요건과 무관하다.

> **참고 (informative)**
>
> 가) 다) 는 오늘날 쓰이는 거의 모든 기계가 이미 만족한다. 이 문서가 그것을 **적어 두는** 이유는, 적어 두지 않으면 「이 언어는 어디서 도는가」가 소스 밖의 지식이 되기 때문이다. 바이트가 8 비트가 아닌 일부 신호처리 장치는 이 언어의 대상이 아니다.

5.4 진단

1 처리기는 적합하지 않은 프로그램에 대하여 진단 (diagnostic) 을 내야 한다.

2 진단은 **안정된 식별자**를 가져야 한다. 식별자는 대문자와 붙임표로 이루어지며 (예: `E-TYPE-INSTANCE`), 한 번 뜻이 정해진 식별자의 뜻은 바뀌지 아니한다.

3 진단은 그 잘못이 일어난 **자리**(파일·줄)를 알려야 한다.

4 진단의 식별자는 다음 갈래를 가진다.

표 4 — 진단 식별자의 갈래

앞자리	뜻	예
`E-`	잘못. 번역이 성공하지 아니한다	`E-TYPE-INSTANCE`, `E-EFFECT`
`W-`	경고. 번역은 되지만 저자가 봐야 한다	`W-EXPORT-NOSYM`
`E-VM-`	실행 중 트랩	`E-VM-CONTRACT`, `E-VM-BOUNDS`

5 실행 중 트랩은 프로그램을 즉시 멈추며, 그 사실을 실행 환경에 알린다. 트랩 뒤에 계산이 계속되지 아니한다.

> **주의 — 진단 메시지의 글은 계약이 아니다**
>
> 계약인 것은 **식별자**이고, 사람이 읽는 문장은 편의다. 도구를 만드는 사람은 문장을 정규식으로 뜯지 말고 식별자를 봐야 한다 — 문장은 다듬어질 수 있다.

5.5 프로그램의 시작과 끝

1 **실행되는 프로그램**에는 시작점 (entry point) 이 정확히 하나 있어야 한다. 시작점은 `main` 이라는 이름의 op 이다.

1a 시작점이 없는 번역 단위도 적합하다 — 그것은 다른 프로그램이 가져다 쓰는 라이브러리 모듈 (library module) 이다. 시작점이 필요한 때는 실행할 수 있는 결과물을 만들 때다.

2 시작점이 돌려주는 값이 프로그램의 종료 상태 (exit status) 다. 0 은 성공을 뜻하고, 그 밖의 값의 뜻은 실행 환경이 정한다(구현 정의).

2a 시작점의 출력으로 쓸 수 있는 것은 다음이다 — 작은 부호 없는 정수(`u8`·`u32` 따위) · `void`(끝나면 성공이다) · `result`(성공이면 0, 오류면 그 오류가 정하는 상태).

2b 값이 실행 환경의 종료 상태 폭보다 넓으면 어떻게 좁히는지는 구현이 정한다.

3 호스티드 환경에서 시작점은 실행 환경이 주는 권한을 인자로 받을 수 있다. 어떤 권한을 받을지는 시작점이 자기 계약에 적으며, 실행 환경은 적힌 것만 건넨다(§7.2).

4 시작점이 돌아오면 프로그램이 끝난다. 끝나기 전에 열려 있던 자원은 소유 규칙 (§8.5)에 따라 이미 닫혀 있어야 하며, 닫히지 않은 자원이 있으면 그것은 번역 단계에서 거부된다.

예제 (example) — 시작점이 권한을 받는다

소스 코드 (source)

```
module ex_env .

use stdio .

rem 안 받은 권한은 이 프로그램 어디에도 없다.
proc main input out cap io . output u8 . effects io .
do
  let n usize stdio.write out. 1 "hello\n" . .
  return cast u8 n. . .
end .
```

거부되는 예제 (rejected) — 시작점은 `u8` 을 돌려준다

소스 코드 (source)

```
module ex_entry .

proc main output u64 . effects none .
do
  return 0 .
end .
```

진단: `E-ENTRY-OUTPUT`

쉬운 말로

마지막 문단이 다른 언어와 크게 다른 자리다. 흔히 「프로그램이 끝나면 운영체제가 알아서 치운다」고 하지만, 로우엔트는 **치우는 것을 잊었다는 사실 자체를 번역 단계에서 거부한다**. 그래서 운영체제가 없는 환경에서도 같은 코드가 성립한다.

5.6 기계의 이름

1 처리기는 어떤 기계 (target) 를 위해 짓는지 안다. 기계마다 워드 크기(word size)·바이트 순서(byte order)· 부동소수점 수(floating-point number)의 유무 따위가 다르며, 그 차이가 프로그램의 뜻에 닿는 자리에서는 이 문서가 그렇다고 말한다.

기계 (target)

프로그램이 돌 곳. 워드 크기·바이트 순서·부동소수점 수의 유무 따위가 정해져 있으며, 처리기는 짓기 전에 그것을 안다. 번역이 일어나는 곳(§5.1 의 번역 환경)과 같을 수도 다를 수도 있다.

2 기계의 이름은 **닫힌 집합**이며 다음이 전부다.

표 5 — 기계 이름

이름	워드(비트)	바이트 순서	비고
`x86_64`	64	리틀 엔디언(little-endian)	부동소수점 수 있음 · 벡터 있음
`arm64`	64	리틀 엔디언	부동소수점 수 있음 · 벡터 있음
`riscv64`	64	리틀 엔디언	부동소수점 수 있음 · 벡터는 고르는 것이라 바탕에 없음
`mips_be`	32	**빅 엔디언(big-endian)**	부동소수점 수 있음

`cortex_m`	32	리틀 엔디언	**힙(heap) 없음 · 부동소수점 수 없음** — 홀로 도는 기계
`cortex_m0`	32	리틀 엔디언	`cortex_m` 과 같고 곱셈기의 폭만 16 비트 (⟦§5.3⟧ (4c)) — Cortex-M0 · M0+
`win64`	64	리틀 엔디언	건너가기만 한다(⟦§5.6⟧ (4))
`cheriot`	32	리틀 엔디언	**포인터가 능력이다**(8 바이트, ⟦§5.3⟧ (4a)) · 힙 없음 · 부동소수점 수 없음 · 건너가기만 한다 (⟦§5.6⟧ (6))
`atmega328p`	16	리틀 엔디언	AVR(Arduino Uno · Nano) · 정렬 1 바이트 · 곱셈기의 폭 8 비트(⟦§5.3⟧ (4c)) · 힙 없음 · 부동소수점 수 없음 · 내고 돌린다(⟦§5.6⟧ (8))
`atmega2560`	16	리틀 엔디언	AVR(Arduino Mega 2560) · `atmega328p` 와 같은 값 · 내고 돌린다(⟦§5.6⟧ (8))
`atmega32u4`	16	리틀 엔디언	AVR(Arduino Leonardo · Micro) · `atmega328p` 와 같은 값 · 짓는 데까지(⟦§5.6⟧ (8))
`atmega4809`	16	리틀 엔디언	AVR(Arduino Nano Every) · `atmega328p` 와 같은 값 · 짓는 데까지(⟦§5.6⟧ (8))
`word16`	16	리틀 엔디언	힙 없음 · 부동소수점 수 없음 · **번역까지만 선다**(⟦§5.6⟧ (7))

2a 기계의 이름들은 **나무**를 이룬다. 마디는 부모의 값을 물려받고 자기가 바꾸는 값만 적는다 — `cortex_m0` 은 `cortex_m` 에서 곱셈기의 폭만 바꾸고, `atmega328p` · `atmega2560` · `atmega32u4` · `atmega4809` 는 묶음 `avr` 의 값을 그대로 물려받는다. 나무의 가지에만 서는 **묶음**(`lowent` · `riscv32` · `avr`)은 기계의 이름이 아니므로 (3) 에 따라 거부된다. 처리기는 마디마다 펼친 값을 보여 준다.

3 이 목록에 없는 이름은 어떤 빌드에도 맞지 아니하므로 거부된다. 맞지 않는 이름을 받아 주면 그 op 은 **조용히 빠지거나 조용히 잘못 지어진다** — 둘 다 이 문서가 막으려는 것이다.

4 기계에 **없는 것은 쓸 수 없다.** 처리기는 그것을 번역할 때 거절한다.

표 6 — 기계가 갖지 못한 것 — 그 자리에서 거절된다

무엇이 없나	어느 기계	쓰면
자라는 뿌리(힙)	`cortex_m` · `cheriot` · `atmega328p` · `atmega2560` · `atmega32u4` · `atmega4809` · `word16`	`heap` 효과. `cap heap.region<이름>heap`

		이거 절된다 (E-HEAP-NOHOST). 고정창에서 깎는 alloc 은 쓸 수 있다 ([[§8.1]])
64 비트 부동소수점 수(f64)	cortex_m · cheriot · atmega328p · atmega2560 · atmega32u4 · atmega4809 · word16	f64 와 타입 낱말 없는 부동 리터럴 이거 절된다 (E-FLOAT-NOFLOAT)

4-1 `f32` 는 **어느 대상에나 있다.** 부동소수점 장치가 없는 대상에서는 소프트웨어로 셈하며, 그 결과는 장치가 있는 대상과 비트까지 같다. `f64` 는 장치가 있는 대상에만 있다 — 위 표의 대상에서는 쓸 수 없다.

4-2 처리기는 소프트웨어 `f32` 를 장치 없는 모든 대상에서 받는다. AVR 넷(`atmega328p` · `atmega2560` · `atmega32u4` · `atmega4809`)에서는 실행해 비트까지 재어 보았고, `cortex_m` · `cheriot` · `word16` 에서는 지었으나 실행해 보지 않았다 — 그 대상에서 `f32` 를 쓰는 선언은 받되, 번역할 때 경고한다(`W-FLOAT-UNTESTED`: 제대로 동작하지 않을 수 있으니 답이 이상하면 개발자에게 알리라). 이 경고는 언어의 규칙이 아니라 처리기가 아직 재지 못했다는 말이다. 소프트웨어 `f32` 를 아직 짓지 못한 대상이 생기면 그 대상은 `f32` 를 거절한다(`E-FLOAT-NOTYET`) — 이 판에는 그런 대상이 없다.

> **쉬운 말로**
>
> `f32` 와 `f64` 를 가른 까닭: 32 비트 부동은 작은 기계에서도 소프트웨어로 정확히 셈할 수 있고, 센서 값 · 제어 계수처럼 그 자리에서 필요한 일이 많다. 64 비트는 셈이 몇 배 무겁고, 작은 기계의 C 컴파일러가 `double` 을 32 비트로 두기도 한다 — 정직하게 낼 수 없는 값을 받는 것보다 거절하는 편이 낫다.

5 기계가 **가진 것**을 쓸지는 짓는 사람이 정한다. 빌드는 그 바이너리가 **담을 명령 집합**을 고르고, 담은 것이 둘 이상이면 프로그램이 **시작할 때 한 번** 골라 고정한다. 실행 도중에는 바뀌지 아니한다 — 무엇이 돌았는지가 실행마다 달라지면 잰 수를 다시 낼 수 없다.

표 7 — 명령 집합의 범위 — 빌드가 고른다

고른 것	무엇이 일어나나
아무것도 안 담음	모든 셈을 평범한 코드로 한다. 어느 기계에서나 선다
집합 하나 이상을 담음	그 명령이 **있다고 보고** 낸다. 없는 기계에서는 돌지 않는다
담되 «고르게» 함	둘을 다 담고 시작할 때 한 번 판별해 고정한다. 어느 기계에서나 서고, 있는 기계에서는 빠르다

5a 어느 쪽을 고르든 **답은 같다**. 기계 명령과 평범한 코드는 같은 뜻의 두 구현이며, 다른 것은 빠르기와 **타이밍 성질**뿐이다 — 표를 읽는 셈은 읽는 자리가 값에 달렸고, 그 명령들은 그렇지 아니하다. 상수시간을 약속하는 자리는 그 차이를 적는다(§9.2).

5c 담는 것은 **셈을 대신하는 명령**만이 아니다 — **폭**(한 레지스터에 서로 독립인 일 여럿을 나란히 싣는 자리)도 같은 범위에 든다. 규칙은 하나로 같다: 답은 같고(§5.6 (5a)), 없는 기계에는 담지 못하며(§5.6 (5b)), 고르는 것은 **시작할 때 한 번**이다.

5b 기계에 그 명령이 없으면 담을 수 없다. 담으라고 적으면 번역이 거절된다 — 조용히 평범한 코드로 바꾸지 아니한다. 무엇이 돌지 짓는 사람이 알아야 한다.

4a 곧 «임베디드에서도 도는 코드» 는 **바라는 것이 아니라 재는 것**이다 — 자라는 뿌리나 부동소수점 수를 쓴 프로그램은 그 기계를 고르는 순간 번역이 멈춘다.

> **쉬운 말로**
>
> 왜 실행할 때가 아니라 **번역할 때** 막는가. 힙이 없는 기계에 할당을 실어 보내면 그것은 늦게, 그 기계 위에서, 고치기 가장 어려운 자리에서 터진다. 없는 것은 **없다고 미리** 말하는 편이 낫다 — 그것이 이 언어가 §1.3 에서 «숨은 비용» 이라 부르는 것의 반대다.

5d `win64` 는 **건너가는 데까지만** 뜻이 있다. 그 기계를 위한 씨 소스를 낼 수 있고 그것이 지어지는 것까지가 이 문서가 말하는 바이며, 그 위에서 도는 것은 말하지 아니한다. POSIX 에만 있는 것을 쓰는 op 은 번역할 때 거부된다 — 뒤에서 헤더가 없다고 터지는 것보다 **이유를 아는 자리에서 먼저 우는 편**이 낫다.

6 `cheriot` 도 **건너가는 데까지만** 뜻이 있다 — 그 기계를 위한 씨 소스를 내고 그것이 지어지는 것까지다. 이 기계의 포인터는 능력이며(§5.3 (4a)), `machine.ptr_caps` 가 참이다. 레지스터 묶음의 포인터는 실행 환경이 주는 장치 메모리의 능력에서 파생된다(§7.6.1 (3)).

7 `word16` 은 **번역하는 데까지만** 뜻이 있다. 실제 기계의 이름이 아니라 `usize` · `idiff` 의 폭이 16 비트인 구현(§6.2 (3-1))에서 같은 글이 서는지를 재는 자리다. 처리기는 이 대상으로 글을 번역하고 검사하되, 씨 소스를 내지도 그 글을 돌리지도 아니한다 — 그 폭에서 낸 것이 옳은지 잰 적이 없기 때문이다. 슬라이스 길이의 상한은 2^{15} 이다.

8 `atmega328p` 와 `atmega2560` 은 씨 소스를 내고 돌리는 데까지 뜻이 있다. 그 기계를 위해 낸 씨 소스가 그 칩의 씨 컴파일러로 지어지고, 그것을 그 칩을 흉내 내는 기계에서 돌린 답이 같은 글의 다른 실행과 같음을 잰 데까지다. `atmega32u4` 와 `atmega4809` 는 **짓는 데까지** 뜻이 있다 — 흉내 내는 기계가 없어 돌린 답을 재지 못했다. 씨 `int` 는 16 비트이므로 처리기는 32 비트 낱말을 폭이 정해진 씨 타입으로 낸다. 원자 연산은 인터럽트를 막은 동안의 읽고-쓰기로 낸다 — 이 칩들은 코어가 하나다. 슬라이스 길이의 상한은 2^{15} 이다.

5.7 이 단위가 무엇인지 밝히는 자리 — `package`

1 번역 단위는 자기가 **무엇인지**를 `package <열쇠말> <값> .` 으로 적을 수 있다. 이것은 빌드 선언(`build tier t0 .` 등)과 **같은 모양**이다 — 관용구가 둘이면 그것도 두 번째 표현이므로, 새 짜임을 만들지 아니한다.

2 열쇠말은 닫힌 집합이며 여섯이다. 그 밖의 이름은 거부된다(`E-PKG-KEY`).

표 8 — `package` 의 열쇠말

열쇠말	있어야 하나	무엇을 적나
`name`	예	이 단위의 이름
`version`	예	판 번호. **차례를 매길 수 있는** 꼴이어야 한다
`description`	아니오	한 줄 설명
`date`	아니오	날짜
`authors`	아니오	지은이
`license`	아니오	쓰임새를 정하는 조건

3 열쇠말을 하나라도 적었다면 `name` 과 `version` 은 **반드시** 있어야 한다 (`E-PKG-NONAME` · `E-PKG-NOVERSION`). 정체를 적는 자리에 정체가 없으면 장식이다.

4 같은 열쇠말을 두 번 적는 것은 거부된다(`E-PKG-DUP`). 정체는 값이 **하나**다 — 둘이면 이 자리를 읽는 도구마다 고르게 되고, 모두가 같은 것을 고르지는 아니한다.

5 `version` 은 **차례를 매길 수 있는 꼴**이어야 하며, 아니면 거부된다 (`E-PKG-VERSION`). 아무도 해석하지 못하는 판 번호는 판을 **차례짓지** 못하고, 차례짓는 것이 판 번호가 있는 유일한 까닭이다.

6 모양이 `package <열쇠말> <값> .` 이 아니면 거부된다(`E-PKG-FORM`).

> **쉬운 말로**
>
> 이 자리가 있는 까닭은 도구와 사람이 **소스 전체를 읽지 않고** 이 단위의 정체를 싸게 알기 위함이다. 그런데 읽어서 믿을 수 없다면 그 값은 통째로 없는 것과 같다 — 그래서 여기만은 **적은 것을 검사한다.** 아무도 읽지 않는 열쇠말은 아무도 검사하지 않는 열쇠말이고, 검사되지 않는 중복은 거짓말로 썩는다(§1.1).

5.8 빌드마다 다르게 짓기 — `build option` 과 `config`

1 프로그램은 **손잡이**를 선언할 수 있다. `build option <이름> <갈래> …` 이 손잡이를 만들고, 그 값은 짓는 자리에서 정해진다.

2 갈래는 셋이며, 그 밖은 거부된다(`E-OPT-TYPE`).

표 9 — 손잡이의 갈래

갈래	무엇을 담나
`bool`	켜짐과 꺼짐
`int`	정수
`choice`	적어 둔 값들 가운데 하나

3 손잡이는 다른 손잡이에 **기댈** 수 있다(`depends`). 기대는 쪽이 `bool` 이고 **켜져** 있는데 기댐을 받는 쪽이 꺼져 있으면 거부된다(`E-CONFIG-DEPENDS`) — **그런 빌드는 없기** 때문이며, 이 검사가 없으면 `depends` 는 장식이 된다.

3a 기대는 쪽이 `choice` 나 `int` 이면 「켜짐」 이라는 자리가 없다 — 그 손잡이는 늘 값을 지닌다. 그때 기댐이 안 맞는 것은 **없는 빌드가 아니라 쓰이지 않는 손잡이**이며, 구성이 굳이 값을 고르면 알린다(`W-CONFIG-DEPENDS`).

> **쉬운 말로**
>
> (3a) 이전에는 「켜짐」 을 값이 0 이 아닌 것으로만 재서, 0 을 고를 수 없는 `choice` 손잡이가 **늘 켜진 것**이 됐다. 그래서 그것이 기대는 손잡이를 끄는 구성이 **하나도 존재할 수 없었다** — 어떤 구성으로도 끌 수 없는 손잡이는 손잡이가 아니다(결함 노트 18).

4 손잡이의 값은 `config <이름>` 으로 읽으며, 그것은 **번역 시점 상수**다(§6.8). 없는 이름을 읽으면 거부된다(`E-CONFIG-UNDEF`).

4a 기댐을 받는 손잡이가 아예 **없으면** 거부된다(`E-OPT-DEPENDS`).

4b 구성이 손잡이에 **그 갈래가 받을 수 없는 값**을 주면 거부된다(`E-CONFIG-TYPE`). 모르는 성질을 적는 것도 그렇다(`E-CFG-UNKNOWN-PROP`).

5 선언해 놓고 **아무 코드도 읽지 않는** 손잡이는 거부된다(`E-OPT-UNUSED`). 그런 손잡이는 구성에 나타나고, 쓰는 사람이 그것을 끄고, 그런데 아무 일도 일어나지 않는다 — **결정처럼 보이지만 결정이 아닌 것**이다.

6 값이 정해지면 **꺼진 가지는 생성물에 남지 아니한다.** 비용이 0 이다.

> **주의 — 꺼진 가지도 검사는 받는다**
>
> 다른 언어의 조건부 컴파일에서는 안 켜진 조합의 코드가 아무에게도 읽히지 않은 채 썩는다. 여기서는 그럴 수 없다 — 접히는 것은 **코드 내기**이지 파싱과 타입 검사가 아니다. 그래서 「그 조합에서는 빌드가 안 된다」 는 일이 생기지 아니한다.

5.9 정했으되 짓지 않은 도구 — 검증된 판 · 패키지 파일 · 사설 원천 · 큰 빌드의 나눔

1 이 조항은 처리기 둘레의 **도구**가 할 일 가운데 뜻이 정해져 있되 아직 짓지 않은 것을 적는다. 언어의 낱말이 아니므로 거부할 낱말은 없다. §6.9.4 와 같이 쓰임이 나올 때 짓고, 필요 없다고 보이면 버린다. 짓을지 버릴지는 소유자가 정한다.

2 **검증된 판 번호** — 판 번호의 갈래(호환이 깨지는 판 · 깨지지 않는 판)를 저자가 매기는 것이 아니라, 도구가 그 단위의 **드러난 겉** (서명 · 계약 · 효과 · 배치)의 차이로 정한다. 겉이 깨지는데 깨지지 않는 판으로 매기면 내놓기가 거부된다. 의존을 범위로 적으면(호환되는 판 아무것) 도구가 호환되는 가장 새 판을 고르고, 깨지는 판으로 옮길 때는 사람에게 묻는다. 범위를 적는 철자는 짓을 때 정한다. 지금 서 있는 것: 판 번호는 차례를 매길 수 있어야 한다(§5.7 (5)), 그리고 잠금 파일(`lowent.lock`)이 의존의 내용 해시를 고정한다.

3 **패키지 파일**(`.lowpkg`) — 매니페스트 · 소스 · 메타 · 서명을 한 덩이로 묶어 내놓는 파일. 새 암호 형식을 짓지 않고 검증된 묶음 위의 얇은 관례로 둔다. 지금 서 있는 것: 소스를 경로로 받는 의존 · 내용 해시 · 서명과 신뢰 키(`lowentc key` · `lowentc verify`) · 미러.

4 **사설 원천의 인증** — 사설 서버에서 패키지를 받을 때 자기를 밝히는 일. 지금 서 있는 것: 받는 원천의 정책(어디서든 · 이 기계에서만 · 공식 키로 서명된 것만)과 내용 해시 — 어디서 받았든 해시가 같으면 같은 패키지다.

5 **큰 빌드의 나눔** — 빌드 정보를 뜻에 따라 조각으로 나누고(조각의 쓰임 · 크기 한도 · 묶음의 구성원), 손잡이(§5.8)의 값을 고르는 화면 도구(`lowent config`)를 두는 일. 지금 서 있는 것: 손잡이와 `config`, 그리고 `pkg.low`.

> **참고 (informative)**
>
> 출처는 RFC-0023(검증된 판 · 의존의 범위) · RFC-0032 · 0033(원천 · 인증) · RFC-0034(`.lowpkg`) · RFC-0036 D1 D4(큰 빌드의 나눔)다. 2026-11-11 의 훑기(F7)는 RC11 메모리 순서(RFC-0018) · `match` 의 결정 트리(RFC-0081 MM9) · 서명 · 미러 · 손잡이도 «짓지 않음» 으로 적었으나 그것은 **틀렸다** — 그 다섯은 지어져 있다(§7 의 차례 다섯 · 점프 표 · `lowentc verify` · 미러 설정 · §5.8). 2026-11-12 에 바로잡았다. 소유자는 §6.9.4 와 같이 «정식으로 넣되, 차후 불필요하게 되면 버린다고 명시해서 계속 추적» 하기로 했다(2026-11-12).

6 언어 (Language)

1 이 조항은 로우엔트의 문법과 의미를 정한다. 먼저 소스가 어떤 조각으로 나뉘는지를 정하고 (§6.1), 그 조각들이 무엇을 뜻하는지를 차례로 정한다.

6.1 어휘 (Lexical elements)

구문

```
source    ::= { separator ¦ token }
separator ::= whitespace ¦ line-comment ¦ block-comment
token     ::= keyword ¦ name ¦ path ¦ literal ¦ punctuation

punctuation ::= "." ¦ "(" ¦ ")" ¦ "," ¦ operator
operator    ::= "+" ¦ "-" ¦ "*" ¦ "/"
```

정적 의미

1 처리기는 소스 파일을 앞에서부터 읽어 `separator` 와 토큰 (token) 의 나열로 가른다. 토큰은 다섯 갈래다 — 낱말(`keyword`), 이름(`name`), 경로(`path`, §6.1.3), 리터럴(`literal`, §6.1.4), 구두점(`punctuation`).

2 `separator` 는 토큰을 가르고 **버려진다.** 그 밖의 뜻은 없다.

2a 구두점 가운데 떨어진 점 `.` 은 폼을 닫고 `( )` 는 폼을 묶는다(§6.1.5 · §6.1.6). `operator` 는 `expr` 섬 안에서만 연산자다(§6.3.2).

3 영문자 · 숫자 · 밑줄이 이어진 것은 **끊지 아니하고 한 덩이로** 읽은 뒤에 그 갈래를 정한다. `iffy` 는 낱말 `if` 로 시작하는 것이 아니라 이름 하나이고, `remark` 는 주석이 아니라 이름이다.

6.1.1 공백과 주석

구문

```
whitespace      ::= " " ¦ tab ¦ carriage-return ¦ newline
blank           ::= " " ¦ tab ¦ carriage-return
tab             ::= ? U+0009 ?
carriage-return ::= ? U+000D ?
newline         ::= ? U+000A ?
line-comment    ::= "rem" { line-char }
block-comment   ::= "note" blank { blank } tag { line-char } newline
                    { comment-line newline }
                    tag { blank } ( newline ¦ end-of-source )
tag             ::= name
line-char       ::= ? newline 이 아닌 글자 ?
comment-line    ::= ? 닫는 줄이 아닌 줄 ?
end-of-source   ::= ? 소스의 끝 ?
```

1 `block-comment` 의 **닫는 줄**은 줄의 맨 첫 칸에서 여는 줄의 `tag` 와 같은 글자열로 시작하고, 그 바로 뒤가 `name-char`(§6.1.3) 가 아닌 줄이다. 그 밖의 줄은 `comment-line` 이다 — 들여 쓴 `tag` 와 `tag` 보다 긴 낱말로 시작하는 줄이 그렇다.

제약

2 `rem` 과 `note` 는 그 바로 뒤가 `name-char` 가 아닐 때에만 주석을 연다 — `remark` 와 `notes` 는 이름이다 (§6.1 (3)).

3 `note` 다음에는 `tag` 가 있어야 한다.

4 `block-comment` 를 닫는 `tag` 뒤에는 `blank` 와 `newline` 만 올 수 있다.

5 열린 `block-comment` 는 소스가 끝나기 전에 닫는 줄을 만나야 한다.

정적 의미

6 공백(스페이스 · 탭 · 캐리지 리턴 · 줄바꿈)은 토큰을 가른다. 토큰을 가르는 것 말고 다른 뜻은 없다.

7 `rem` 으로 시작하는 자리부터 그 줄의 끝까지는 주석 (comment) 이며, 처리기는 그것을 없는 것으로 다룬다. `block-comment` 도 주석이다 — 여는 줄부터 닫는 줄까지를 없는 것으로 다룬다.

8 주석 안에는 유니코드 문자를 쓸 수 있다.

9 `rem` 과 `note` 는 이름이 아니다. 이 둘로 시작하는 자리는 언제나 주석이다.

진단

표 10 — 주석의 진단

어긴 것	진단
(3) — `note` 뒤에 태그가 없다	`E-NOTE-TERM`
(4) — 닫는 태그 뒤에 다른 글자가 있다	`E-NOTE-TAIL`
(5) — 닫는 줄 없이 소스가 끝났다	`E-NOTE-UNTERM`

예제

예제 (example) — 주석

소스 코드 (source)

```
rem 이 줄은 전부 주석이다. 한글도 쓸 수 있다.
let n u32 42 .   rem 여기서부터 줄 끝까지도 주석이다.
```

예제 (example) — 여러 줄 주석

소스 코드 (source)

```
module ex_note .

note DOC
여러 줄에 걸친 설명이다.
처리기는 이것을 없는 것으로 다룬다.
DOC

fn f output u8 . do return 1 . end .
```

실행 결과 (output)

```
f() = 1
```

참고

참고 (informative)

주석을 여는 낱말이 `rem` 인 것은 그것이 **낱말**이기 때문이다. 이 언어에는 기호로 된 주석 표시(`//`·`/* */`)가 없다 — 어휘를 낱말 하나로 통일하면 읽는 규칙이 하나로 줄어든다.

참고 (informative)

`block-comment` 는 텍스트 리터럴(§6.1.4)과 같은 모양이되 **값이 아니라 주석**이다. 닫는 줄의 규칙이 같다. 한 줄짜리 주석은 `rem` 으로 적는다.

6.1.2 낱말 (Keywords)

구문

```
keyword ::= "actor" ¦ "break" ¦ "case" ¦ "continue" ¦ "contract" ¦ "cycle" ¦ "do"
          ¦ "drop" ¦ "else" ¦ "end" ¦ "enum" ¦ "expect" ¦ "export" ¦ "expr"
          ¦ "extern" ¦ "false" ¦ "fn" ¦ "for" ¦ "guard" ¦ "if" ¦ "keep"
          ¦ "let" ¦ "lit" ¦ "match" ¦ "module" ¦ "newtype" ¦ "next" ¦ "none"
          ¦ "proc" ¦ "range" ¦ "repeat" ¦ "return" ¦ "satisfies" ¦ "send" ¦ "set"
          ¦ "spawn" ¦ "state" ¦ "step" ¦ "struct" ¦ "test" ¦ "trait" ¦ "true"
          ¦ "try" ¦ "type" ¦ "unsafe" ¦ "use" ¦ "var" ¦ "while"
```

제약

1 낱말 (keyword) 은 언어가 뜻을 정해 둔 이름이며, 다른 뜻으로 쓸 수 없다. 낱말과 같은 철자는 이름이 될 수 없다.

2 낱말의 목록은 고정되어 있다. 위의 생성 규칙이 그 전부이며 부록 A 의 목록과 같다.

2a 예약만 해 두고 쓰지 아니하는 낱말은 없다. 낱말마다 그것을 쓰는 구문이 이 문서에 있다.

3 타입 선언은 제 낱말로 시작한다 — `struct` · `enum` · `type` · `newtype`. 다른 선언(`fn` · `proc` · `actor` · `trait` · `test` · `module`)과 같이 앞에 붙는 낱말이 없다.

4 `def` 는 낱말이 아니다. 저자의 이름이 될 수 있다. 다만 옛 머리 `def struct` · `def enum` · `def type` · `def newtype` 은 그 꼴로 알아보고 거부한다.

5 `export` 는 선언의 낱말 앞에 온다: `export struct rect do … end .`.

진단

표 11 — 낱말의 진단

어긴 것	진단
(1) — 낱말을 이름으로 지었다	`E-NAME-KEYWORD`
(4) — 타입 선언을 `def` 로 시작했다	`E-DEF-OLD`

예제

거부되는 예제 (rejected) — 타입 선언을 옛 머리 `def` 로 시작했다

소스 코드 (source)

```
module ex_defold .

def struct pt do
  x u32 .
end .
```

진단: `E-DEF-OLD`

예제 (example) — `def` 는 저자의 이름이 될 수 있다

소스 코드 (source)

```
module ex_defname .

struct pt do
  x u32 .
end .

fn twice input def u64 . output u64 . do
  return add def. def. . .
end .
```

실행 결과 (output)

```
twice(4) = 8
```

참고

> **참고 (informative)**
>
> 낱말은 **예산**이다. 새 낱말을 더하는 것은 언어가 커지는 일이므로, 기존 낱말로 표현할 수 있는 것에는 새 낱말을 주지 아니한다.

> **쉬운 말로**
>
> 「예산」이라는 말이 낯설 수 있다. 뜻은 이렇다 — 낱말 수를 세어서 관리하고, 하나 늘릴 때마다 그만한 값어치가 있는지 따진다는 것이다. 이 언어의 낱말 수는 사람이 한 화면에 담아 외울 수 있는 크기로 유지된다.

6.1.3 이름 (Identifiers)

구문

```
name       ::= name-start { name-char }
name-start ::= "a" … "z" ¦ "A" … "Z" ¦ "_"
name-char  ::= name-start ¦ "0" … "9"
path       ::= name "." path-part { "." path-part }
path-part  ::= name-char { name-char }
```

1 `path` 의 점은 앞뒤 글자에 **붙어** 있다. 빈칸으로 떨어진 점은 `path` 의 일부가 아니라 구두점이다 (§6.1.5).

제약

2 `name` 은 낱말(§6.1.2)과 같은 철자일 수 없다.

3 이름에 유니코드 문자를 쓸 수 없다(§5.2). 이름을 이루는 글자는 위의 생성 규칙이 정한 ASCII 글자뿐이다.

4 **짓는 이름에는 점이 없다.** 점이 붙은 이름을 선언할 수 없다.

5 기본 연산의 이름(§6.3.3)과 같은 철자를 매개변수나 지역 이름으로 쓸 수 없다. 빌트인 연산의 이름은 제 이름공간에 있으므로 이 규칙에 들지 아니한다.

정적 의미

6 이름 (identifier) 은 저자가 짓는 이름이다. 모듈 · op · 타입 · 변수가 이름을 갖는다.

7 점은 **가리킬 때만** 나타난다. `path` 는 **선언된 이름을 가리키는 경로**이며 그 뜻은 §6.1.5 (5) 가 정한 셋뿐이다 — 모듈의 이름, 변형의 이름, 타입에 딸린 선언의 이름. 점은 이름의 일부가 아니라 **이름들을 잇는 표시**다.

8 값 안의 필드를 고르는 것은 `path` 로 적지 아니한다. 그것은 `field` 폼이다(§6.2.10).

진단

표 12 — 이름의 진단

어긴 것	진단
(2) — 낱말을 이름으로 지었다	`E-NAME-KEYWORD`
(3) — 이름에 ASCII 가 아닌 글자가 있다	`E-NAME-ASCII`
(4) — 점이 붙은 이름을 선언했다	`E-NAME-DOTTED`
(5) — 기본 연산의 이름을 지역 이름으로 지었다	`E-NAME-BUILTIN`
(8) — 값의 필드를 붙임 점으로 적었다(`p.x`)	`E-FIELD-GLUED`

예제

> **거부되는 예제 (rejected) — 점이 붙은 이름은 지을 수 없다**
>
> **소스 코드 (source)**
>
> ```
> module ex_dotted .
>
> fn f output u8 .
> do
>   let a.b u8 1 .
>   return 1 .
> end .
> ```
>
> 진단: `E-NAME-DOTTED`

참고

> **쉬운 말로**
>
> 여러 겹의 이름 공간은 이 언어에 **없다.** 이름 안에 계층을 담고 싶으면 밑줄로 적는다 (`http_header_parse`). 점을 이름에 허용하면 `a.b` 가 *"a 의 필드 b"* 인지 *"a.b 라는 이름"* 인지 읽는 사람이 알 수 없게 된다 — 그것이 곧 의미 엔트로피다(§1.3).

> **쉬운 말로**
>
> 낱말을 이름으로 쓰면 **선언한 그 자리에서** 거부된다 — 쓰기를 기다리지 않는다. 진단은 어느 이름이 문제인지 짚는다.

> **주의 — 이름이 같으면 헷갈리는 것이 아니라 틀린다**
>
> 어떤 언어는 안쪽 이름이 바깥 이름을 **가리도록**(shadowing) 허용한다. 가려진 이름은 소스를 읽은 사람이 **다른 값을 떠올리게** 만들고, 그것이 곧 의미 엔트로피다 (§1.3). 그래서 이 언어는 가림을 **전면 금지한다** — 기본 연산의 이름, 모듈에 있는 이름, 매개변수의 이름, 그리고 바깥 블록에 살아 있는 이름 중 어느 것도 가릴 수 없다(§6.4.8).

6.1.4 리터럴 (Literals)

구문

```
literal        ::= number ¦ char-literal ¦ string-literal ¦ text-literal

number         ::= [ sign ] ( float ¦ hex ¦ bin ¦ dec )
sign           ::= "+" ¦ "-"
digit          ::= "0" … "9"
hex-digit      ::= digit ¦ "a" … "f" ¦ "A" … "F"
bin-digit      ::= "0" ¦ "1"
dec            ::= digit { [ "_" ] digit }
hex            ::= "0" ( "x" ¦ "X" ) hex-digit { [ "_" ] hex-digit }
bin            ::= "0" ( "b" ¦ "B" ) bin-digit { [ "_" ] bin-digit }
float          ::= dec "." dec [ dec-exponent ]
                 ¦ dec dec-exponent
                 ¦ hex [ "." hex-digit { [ "_" ] hex-digit } ] hex-exponent
dec-exponent   ::= ( "e" ¦ "E" ) [ sign ] dec
hex-exponent   ::= ( "p" ¦ "P" ) [ sign ] dec

char-literal   ::= [ prefix ] single-quote ( char-char ¦ escape ) single-quote
string-literal ::= [ prefix ] double-quote { string-char ¦ escape } double-quote
                 ¦ raw-string
raw-string     ::= "r" double-quote { raw-char } double-quote
raw-char       ::= ? 큰따옴표도 줄바꿈도 아닌 바이트 하나 ?
```

```
prefix         ::= "u" ¦ "U"
escape         ::= backslash ( simple-escape
                             ¦ "x" hex-digit hex-digit
                             ¦ "u" hex-digit hex-digit hex-digit hex-digit
                             ¦ "U" hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-
digit )
simple-escape  ::= backslash ¦ double-quote ¦ single-quote
                 ¦ "a" ¦ "b" ¦ "f" ¦ "n" ¦ "r" ¦ "t" ¦ "v" ¦ "0"
single-quote   ::= ? U+0027 ?
double-quote   ::= ? U+0022 ?
backslash      ::= ? U+005C ?
char-char      ::= ? 작은따옴표도 역슬래시도 줄바꿈도 아닌 글자 하나 ?
string-char    ::= ? 큰따옴표도 역슬래시도 줄바꿈도 아닌 바이트 하나 ?

text-literal   ::= "text" blank { blank } tag [ blank { blank } processor ] { blank } [ line-comment ] newline
                   { text-line newline }
                   tag { blank } ( newline ¦ end-of-source )
processor      ::= "u" ¦ "U"
text-line      ::= ? 닫는 줄이 아닌 줄 ?
```

1 `sign` 은 숫자에 **붙어** 있어야 부호다. `-5` 는 `number` 하나이고 `- 5` 는 아니다. `expr` 섬(§6.3.2) **밖**에서 숫자에 붙은 부호는 값의 일부이고, 섬 **안**에서 `-` 와 `+` 는 연산자다 — 그러므로 폼 자리의 `g 10 -3` 은 인자 둘이다.

2 점도 지수도 없으면 `dec` 로 읽고, 점이나 지수가 있으면 `float` 로 읽는다. 점의 **앞뒤 모두**에 숫자가 있어야 `float` 다 — `.5` 와 `1.` 은 `float` 가 아니며, 그 점은 폼을 닫는 표시로 읽힌다(§6.1.6).

3 `prefix` 는 따옴표에 **붙어** 있어야 접두사다. 사이에 빈칸이 있으면 이름 하나와 리터럴 하나다.

4 `text-literal` 의 **닫는 줄**은 줄의 맨 첫 칸에서 여는 줄의 `tag` 와 같은 글자열로 시작하고, 그 바로 뒤가 `name-char`(§6.1.3) 가 아닌 줄이다. 그 밖의 줄은 `text-line` 이다 — 들여 쓴 `tag` 와, `tag` 로 시작하되 더 긴 낱말(`tag` 가 `DOC` 일 때 `DOCS`)로 시작하는 줄이 그렇다.

제약

5 밑줄은 자릿수 **사이**에만 온다. 수의 맨 앞, 맨 뒤, 진법 표시 바로 뒤, 밑줄 바로 뒤에는 올 수 없다.

6 진법 표시(`0x` · `0b`)와 지수 표시 뒤에는 그 진법의 자릿수가 하나 이상 있어야 한다.

7 `number` 바로 뒤에 영문자나 밑줄이 이어질 수 없다.

8 정수 리터럴의 값은 그것이 쓰이는 자리의 타입 범위 안에 있어야 한다. 부동소수점 리터럴도 그 자리의 타입이 나타낼 수 있는 범위 안에 있어야 한다.

9 부동소수점 타입의 자리에는 `float` 를, 정수 타입의 자리에는 정수 리터럴을 적어야 한다. 둘은 서로 바뀌지 아니한다 — `f64` 자리에는 `3` 이 아니라 `3.0` 을 적는다.

10 `char-literal` 은 접두사가 정한 원소 **하나**에 들어가야 한다. 빈 것도, 원소 하나를 넘는 것도 적을 수 없다.

11 `escape` 는 위의 생성 규칙이 정한 **열넷**뿐이다. 그 밖의 글자를 역슬래시 뒤에 적을 수 없다.

표 13 — 이스케이프 — 닫힌 집합 열넷

적는 것	값	뜻
\\	0x5C	역슬래시 자신
\"	0x22	큰따옴표
\'	0x27	작은따옴표
\a	0x07	경보(bell)
\b	0x08	한 칸 뒤로(backspace)
\f	0x0C	쪽 넘김(form feed)

`\n`	0x0A	줄 바꿈
`\r`	0x0D	줄 처음으로
`\t`	0x09	가로 탭
`\v`	0x0B	세로 탭
`\0`	0x00	영 바이트
`\xNN`	0x00–0xFF	십육진 **두 자리**
`\uXXXX`	코드포인트	십육진 **네 자리**
`\UXXXXXXXX`	코드포인트	십육진 **여덟 자리**

12 `\uXXXX` 와 `\UXXXXXXXX` 가 적는 값은 코드포인트여야 한다. 서러게이트 자리(D800 DFFF)와 10FFFF 를 넘는 값은 적을 수 없다.
13 `prefix` 는 `u` 와 `U` 둘뿐이다. 그 밖의 낱말을 따옴표에 붙여 적을 수 없다. 날 문자열의 `r`(13a)은 큰따옴표에만 붙는다 — `r'…'` 은 없다.

> **날 문자열** (raw string)
>
> 본문을 적힌 그대로 담는 문자열 리터럴. 역슬래시가 이스케이프를 열지 않고, 닫는 큰따옴표에서 끝난다.

13a 날 문자열 (raw string) `r"…"` 의 본문에는 **이스케이프가 없다.** 역슬래시도 그냥 바이트이며, 닫는 큰따옴표에서 리터럴이 끝난다. 그러므로 날 문자열에는 큰따옴표를 넣을 수 없다 — 넣어야 하면 보통 문자열(`\"`)을 쓴다. 원소는 바이트이고(`slice u8`) `u` · `U` 와 함께 쓰지 아니한다. 정규식 · 경로 · 서식 문자열처럼 역슬래시가 많은 글을 적힌 그대로 담는 자리다.
14 `text-literal` 을 닫는 `tag` 뒤에는 `blank` 와 `newline` 만 올 수 있다. 문장을 닫는 점은 **다음 줄**에 적는다.
15 `text-literal` 을 여는 줄에서 `tag` 뒤에는 `processor` 와 주석만 올 수 있다. `text` 다음에는 `tag` 가 있어야 한다.

정적 의미

16 리터럴 (literal) 은 소스에 직접 적은 값이다.
17 정수 리터럴은 십진(`dec`) · 십육진(`hex`) · 이진(`bin`)으로 적는다. 밑줄은 값에 들지 아니한다 (`1_000_000`).
18 **앞의 0 은 팔진이 아니다.** `0755` 는 755 다. 팔진 표기는 이 언어에 없다.
19 정수 리터럴은 **그 자체로는 타입이 없다.** 쓰이는 자리가 타입을 정한다.
19-1 처리기가 번역할 때 정수 리터럴을 담는 정밀도는 **128 비트**다 — 부호 없이 $2^{128}-1$, 부호 있게 -2^{127} 까지. 64 비트(`u64` 의 끝 · `i64` 의 끝)를 넘는 리터럴은 128 비트 자리(§6.2.2 (3b))에만 들어가고, 다른 자리에서는 번역이 거부된다. 128 비트를 넘는 리터럴은 어디서든 거부된다(§6.1.8 (3)).
19a 리터럴로만 된 식 — 리터럴끼리의 셈, 왼쪽이 리터럴인 밀기(`shl 1 n.`) — 도 타입이 없다. 그 식을 받는 자리 (반환 · 묶기 · 셈의 상대편 · 호출의 인자 · `set`)가 정수 타입을 정하면 **그 식은 그 타입에서 셈한다.** 넘치거나 미는 칸 수가 폭 이상이면 다른 셈과 같이 멈춘다(§6.3.4 · §6.3.5). 밀기는 왼쪽만 그 타입을 받는다 — 오른쪽은 미는 칸 수다.
20 부동소수점 리터럴 (floating-point literal) 은 `float` 로 적은 수다 — `1.5` · `1e3` · `1.5e-3`. 십육진 지수 표기의 `p` 뒤는 2 의 거듭제곱이다(`0x1p3` 은 8.0).
21 `inf` 나 `nan` 을 적는 리터럴은 없다. 그런 값은 계산으로 얻는다.
22 참 · 거짓은 낱말 `true` · `false` 로 적는다. 이 둘은 `bool` 타입이며 정수와 서로 바뀌지 아니한다 (§6.2.3).
23 문자 리터럴 (character literal) 의 값은 그 글자의 부호이며 정수다.

24 문자열 리터럴 (string literal) 의 값은 원소의 줄이다. `len` 은 **원소의 수**를 낸다.

25 역슬래시로 시작하는 `escape` 를 이스케이프 (escape) 라 한다. `\xNN` 은 십육진 두 자리를, `\uXXXX` 는 네 자리를, `\UXXXXXXXX` 는 여덟 자리를 **정확히** 먹는다 — 자리 수가 값에 따라 달라지지 아니한다.

26 `\uXXXX` 와 `\UXXXXXXXX` 는 **코드포인트를 적는다.** 그 코드포인트가 어떤 원소로 실리는가는 접두사가 정한다 — 접두사가 없으면 UTF-8 바이트, `u` 면 UTF-16 코드 유닛, `U` 면 코드포인트 그대로다.

27 접두사 (prefix) 는 리터럴의 **원소가 무엇인지**를 정한다. 접두사가 없으면 바이트, `u` 는 UTF-16 코드 유닛, `U` 는 코드포인트다. 문자 리터럴도 같다 — `u'…'` 는 코드 유닛 하나, `U'…'` 는 코드포인트 하나다.

표 14 — 접두사가 정하는 것 — `len` 은 언제나 원소의 수다

적은 것	원소	`len "한" .`	`len "ab" .`	`len "😀" .`
`"…"` (없음)	바이트	3	2	4
`u"…"`	UTF-16 코드 유닛	1	2	2
`U"…"`	코드포인트	1	2	1

28 텍스트 리터럴 (text literal) 은 여러 줄을 그대로 담는 문자열이다. 값은 `text-line` 들을 그 사이의 줄바꿈과 함께 이은 것이다. **닫는 줄 바로 앞의 줄바꿈 하나는 값에 들지 아니한다** — 본문 끝에 줄바꿈을 남기려면 닫는 줄 앞에 빈 줄을 하나 둔다.

29 텍스트 리터럴의 본문에서는 **이스케이프를 풀지 아니한다.** 적힌 바이트가 곧 값이다.

30 `processor` 는 값의 원소를 정한다 — `text TAG u` 는 UTF-16 코드 유닛의 나열(`slice u16`), `text TAG U` 는 코드포인트의 나열(`slice u32`)이다. `processor` 가 없으면 바이트의 나열(`slice u8`)이다.

진단

표 15 — 리터럴의 진단

어긴 것	진단
(5) — 밑줄이 자릿수 사이가 아닌 곳에 있다	`E-NUM-SEP`
(6) — 진법 표시나 지수 표시 뒤에 자릿수가 없다	`E-NUM-EMPTY`
(7) — 수 바로 뒤에 글자가 이어진다	`E-NUM-SUFFIX`
(8) — 리터럴이 그 자리의 타입 범위를 벗어난다	`E-TYPE-WIDTH`
(9) — 부동소수점 자리에 정수 리터럴을 적었다	`E-TYPE-LET`
(19-1) — 64 비트를 넘는 정수 리터럴이 128 비트가 아닌 자리에 있다	`E-LIT-RANGE`
(10) — 문자 리터럴이 비었다	`E-CHAR-EMPTY`
(10) — 문자 리터럴이 원소 하나를 넘는다	`E-CHAR-WIDTH`
(11) · (12) — 이스케이프가 집합 밖이거나 코드포인트가 아니다	`E-STR-ESCAPE`
(13) — 접두사가 `u` · `U` 가 아니거나, `r` 을 작은따옴표에 붙였다	`E-STR-PREFIX`
(14) — 닫는 태그 뒤에 다른 글자가 있다	`E-TEXTLIT-TAIL`
(15) — 여는 줄의 태그 뒤에 처리기도 주석도 아닌 것이 있다	`E-TEXTLIT-OPENER`
(15) — `text` 뒤에 태그가 없다	`E-TEXTLIT-TERM`

예제

예제 (example) — 문자 리터럴

소스 코드 (source)

```
module ex_char .

fn letter output u32 . do return widen u32 'a' . . end .
fn tab    output u32 . do return widen u32 '\t' . . end .
fn quote  output u32 . do return widen u32 '\x27' . . end .
fn hangul output u32 . do return widen u32 U'한' . . end .
fn emoji  output u32 . do return widen u32 U'😀' . . end .
```

실행 결과 (output)

```
letter() = 97 · tab() = 9 · quote() = 39 · hangul() = 54620 · emoji() = 128512
```

예제 (example) — 코드포인트를 적고, 인코딩은 접두사가 정한다

소스 코드 (source)

```
module ex_escape .

fn bytes  output usize . do return len  "\U0001F600" . . end .
fn units  output usize . do return len u"\U0001F600" . . end .
fn points output usize . do return len U"\U0001F600" . . end .

rem 이름이 있는 제어 문자.
fn bell   output u32 . do return widen u32 '\a' . . end .
fn vtab   output u32 . do return widen u32 '\v' . . end .
fn quote  output u32 . do return widen u32 '\'' . . end .
```

실행 결과 (output)

```
bytes() = 4 · units() = 2 · points() = 1 · bell() = 7 · vtab() = 11 · quote() = 39
```

거부되는 예제 (rejected) — 코드포인트가 아닌 것은 적을 수 없다

소스 코드 (source)

```
module ex_surro .

fn f output u32 .
do
  return widen u32 '\uD800' . .
end .
```

진단: E-STR-ESCAPE

거부되는 예제 (rejected) — 팔진 이스케이프는 없다

소스 코드 (source)

```
module ex_octal .

fn f output u32 .
do
  return widen u32 '\101' . .
end .
```

진단: E-STR-ESCAPE

거부되는 예제 (rejected) — 이스케이프 집합 밖은 거부된다

소스 코드 (source)

```
module ex_esc .

fn f output usize .
do
  return len "a\qb" . .
end .
```

진단: `E-STR-ESCAPE`

거부되는 예제 (rejected) — 접두사는 둘뿐이다 — `u8` 은 없다

소스 코드 (source)

```
module ex_pfx .

fn f output usize .
do
  return len u8"ab" . .
end .
```

진단: `E-STR-PREFIX`

예제 (example) — 날 문자열 — 역슬래시가 그대로 남는다

소스 코드 (source)

```
module ex_raw_string .

rem 날 문자열의 `\\` 는 두 바이트다 – 보통 문자열에서는 역슬래시 하나다
fn raw_len output usize .
do
  return len r"\\s192kf%d" . .
end .

fn escaped_len output usize .
do
  return len "\\s192kf%d" . .
end .

rem `r"\n"` 은 줄바꿈이 아니라 역슬래시와 `n` 두 바이트다
fn second output u8 .
do
  return idx r"\n" 1 . .
end .

rem 닫는 큰따옴표 바로 앞의 역슬래시도 그냥 바이트다
fn ends_in_backslash output u8 .
do
  return idx r"a\" 1 . .
end .
```

실행 결과 (output)

```
raw_len() = 10 · escaped_len() = 9 · second() = 110 · ends_in_backslash() = 92
```

거부되는 예제 (rejected) — 날 문자열의 r 은 작은따옴표에 붙지 않는다

소스 코드 (source)

```
module ex_raw_char .

fn f output u8 .
do
  return r'x' .
end .
```

진단: `E-STR-PREFIX`

예제 (example) — 텍스트 리터럴 — 여러 줄을 그대로

소스 코드 (source)

```
module ex_textlit .

fn doc output usize .
do
  return len text DOC
line one
line two
DOC
  . .
end .

rem 본문에서는 이스케이프를 풀지 않는다 — `a\nb` 는 네 바이트다.
fn raw output usize .
do
  return len text RAW
a\nb
RAW
  . .
end .
```

실행 결과 (output)

```
doc() = 17 · raw() = 4
```

예제 (example) — 리터럴

소스 코드 (source)

```
let dec u32 42 .
let hex u32 0x2A .
let big u32 1_000_000 .
let flag bool true .
```

예제 (example) — 부동소수점 리터럴

소스 코드 (source)

```
module ex_float .

fn plain output f64 . do return 1.5 . end .
fn expo  output f64 . do return 1e3 . end .
fn small output f64 . do return 1.5e-3 . end .
```

```
fn under output f64 . do return 1_000.5 . end .
fn hexp  output f64 . do return 0x1p3 . end .
fn negat output f64 . do return -1.5 . end .
```

실행 결과 (output)

```
plain() = 1.5 · expo() = 1000.0 · small() = 0.0015 · under() = 1000.5 · hexp() = 8.0 · negat() = -1.5
```

거부되는 예제 (rejected) — 정수 리터럴은 부동소수점 수가 되지 아니한다

소스 코드 (source)

```
module ex_intfloat .

fn f output f64 .
do
  let x f64 3 .     rem 3 은 정수 리터럴이다 — 3.0 이라고 적어야 한다
  return x. .
end .
```

진단: `E-TYPE-LET`

거부되는 예제 (rejected) — 타입의 범위를 벗어난 리터럴

소스 코드 (source)

```
module ex_lit .

proc p output u8 . effects none .
do
  let x u8 300 .     rem 300 은 u8 의 범위(0~255) 밖이다
  return 0 .
end .
```

진단: `E-TYPE-WIDTH`

참고

쉬운 말로

작은따옴표 자신은 `'\''` 로 적는다(`'\x27'` 도 같은 값이다). 그리고 `u'한'` 이 54620 인 것은 `u` 가 **코드포인트**를 뜻하기 때문이다. 접두사 없이 `'한'` 이라 적으면 그 글자는 바이트 셋이라 한 칸에 안 들어가므로 거부된다.

쉬운 말로

`u"😀"` 이 2 인 것에 주의한다 — 이 글자는 UTF-16 에서 **두 유닛**으로 적히기 때문이다 (서러게이트 쌍). 같은 글자가 `u` 에서는 1 이다. 그래서 "글자 수"라는 말은 이 언어에서 쓰지 아니한다. 언제나 **어느 원소로 세는가**를 먼저 말한다.

쉬운 말로

같은 `\U0001F600` 이 셋 다 다른 수를 낸다 — 그런데 **틀린 것이 하나도 없다.** 이스케이프는 "어떤 글자인가"만 말하고, "몇 조각으로 실리는가"는 접두사가 말한다. 둘을 갈라 두었기 때문에 이스케이프가 인코딩을 몰라도 된다.

> **쉬운 말로**
>
> D800 DFFF 는 UTF-16 이 큰 글자를 두 조각으로 나눌 때 쓰는 **자리**이지 글자가 아니다. 받아 주면 그 리터럴은 아무 글자도 가리키지 않게 된다.

> **쉬운 말로**
>
> 씨(C)에서 `\101` 은 팔진으로 65, 곧 `A` 다. 이 언어에는 팔진 표기가 아예 없으므로 (`0755` 는 755 다) 이스케이프에만 팔진을 되살리지 아니한다 — 그러면 그 자리가 언어에서 **유일한 예외**가 된다.

> **쉬운 말로**
>
> `doc()` 이 17 인 것은 두 줄과 그 사이의 줄바꿈 하나를 더한 값이다 — 마지막 줄 뒤에는 줄바꿈이 붙지 아니한다. `raw()` 가 4 인 것이 "원문 그대로"의 뜻이다: `a`·역슬래시· `n`·`b` 네 바이트이며, 줄바꿈 하나가 아니다.

> **참고 (informative)**
>
> `text` 다음의 태그 자리는 언젠가 **값을 만드는 처리기**의 이름을 받도록 열려 있다 (예컨대 다른 인코딩이나 십육진 바이트열). 이 판의 처리기는 위의 접두사 **둘뿐**이며, 그 밖의 이름은 거부된다. 조용히 통과시키면 `hex"41"` 이 이름 없는 무언가가 되기 때문이다.

> **주의 — 점 앞에 숫자가 없으면 부동소수점 수가 아니다**
>
> 씨(C)·파이썬·자바스크립트는 `.5` 를 0.5 로 읽는다. 로우엔트는 그러지 아니한다 — **앞선 점은 폼을 닫는 표시**이기 때문이다. 닫개와 리터럴이 같은 글자를 다투면 "이 점이 무엇인가"를 앞뒤를 봐야 알게 되고, 그것이 곧 의미 엔트로피다 (§1.3). 그러므로 반드시 `0.5` 라고 적는다. 마찬가지로 `1.` 도 부동소수점 수가 아니며 `1.0` 이라 적는다.

> **주의 — 정수 리터럴이 조용히 잘리지 않는다**
>
> `let x u8 300 .` 은 번역되지 아니한다. 300 은 `u8` 의 범위(0 255) 밖이기 때문이다. 어떤 언어는 이것을 44 로 잘라서 받아들이는데, 그러면 소스에 적힌 300 과 실제 값 44 가 달라진다 — 소스를 읽고도 값을 모르게 되는 것이다.

> **참고 (informative)**
>
> 팔진 이스케이프는 없다. 이 언어에는 팔진 표기 자체가 없기 때문이다((18)).

> **참고 (informative)**
>
> 닫히지 않은 리터럴(따옴표나 닫는 줄 없이 줄이나 소스가 끝난 것)의 진단은 §6.1.7 이 모은다.

6.1.5 점과 form

1 로우엔트의 문법에서 마침표 `.` 는 **닫는 표시**다. 점 하나가 가장 안쪽에 열린 폼 (form) 하나를 닫는다.

2 폼은 전위 표기다 — 이름이 먼저 오고 인자가 뒤에 온다. **쓰는 자리의 이름은 폼을 열고, 점이 그것을 닫는다.** `add a. b. .` 는 `a` 와 `b` 를 더한다: `a.` 와 `b.` 는 인자 없이 곧바로 닫힌 폼, 곧 그 이름의 값이고, 마지막 점이 `add` 를 닫는다.

2a 글자만 보고 값인 줄 아는 것 — 수 · 문자 · 문자열 리터럴과 `true` · `false` · `none` — 은 폼을 열지 아니하며 점을 갖지 아니한다.

2b 다음 자리의 이름은 폼을 열지 아니하며 점을 갖지 아니한다: 이름을 **만드는** 자리(선언의 이름 · `let` · `var` 의 이름 · 입력의 이름 · 되풀이 변수 · 칸의 이름), **타입**의 자리, 그리고 기본 연산과 문장이 정한 **낱말의 자리**(`cast <타입> …` · `field … <마디>` · `send … <처리기>` · 파이프 단계의 낱말 따위). 이 자리들은

부록 A.10 의 문법과 부록 D 의 표가 정한다. 사용자가 선언한 op 의 인자는 **모두 폼**이다 — 타입을 넘길 때도 그렇다(`max_of u64. a. b. .`).

3 폼 안에 폼이 오면 안쪽 폼이 먼저 제 점으로 닫힌다. `add a. mul b. c. . .` 는 `b` 와 `c` 를 곱한 뒤 `a` 를 더한다. 그러므로 폼의 처음과 끝은 **글자만으로** 정해진다 — op 이 인자를 몇 개 받는지 몰라도 나무가 선다. 인자의 수는 나무가 선 뒤에 선언과 견주어 본다(`E-IR-ARITY`).

3a 괄호 `( … )` 는 폼 하나를 감싼다. 뜻을 바꾸지 아니한다 — 읽기 쉽게 묶거나 눈에 띄게 하려고 적는다. 괄호 안의 폼은 제 점으로 닫힌다(`add a. (mul b. c. .) .`). 괄호는 닫개가 아니다.

4 점의 개수는 **검사합**이다 — 열린 폼의 수와 점의 수가 맞지 않으면 처리기가 진단을 낸다. 그래서 점을 빠뜨린 프로그램이 조용히 다른 뜻으로 읽히는 일이 없다.

> **예제 (example) — 전위 표기와 점**
>
> **소스 코드 (source)**
>
> ```
> let total u32 add 1 2 . .
> let mixed u32 add 1 mul 2 3 . . .
> ```

5 이름 **뒤에** 붙은 점(`a.`)은 닫는 점이다 — 띄운 것(`a .`)과 같은 글이다. 이름과 이름 **사이에** 붙은 점은 닫지 않고 한정 (qualification) 을 뜻한다 — 그리고 한정의 뜻은 **셋뿐**이다: 모듈의 이름(`allocs.byte_allocator`), 변형의 이름(`err.too_short`), 그리고 타입에 딸린 선언의 이름(`fn pt.twice`). 이 셋은 모두 **선언된 이름을 가리키는 경로**이지, 값에 대한 연산이 아니다. 이름 **앞에** 붙은 점(`.name`)은 없는 글이다(`E-VOCAB-REMOVED`).

6 값의 안을 들여다보는 것 — 필드 접근과 메서드 호출 — 은 붙임 점으로 적을 수 **없다**. 그것은 폼이며, 다른 모든 폼과 같이 **이름이 먼저 온다**: `field` 와 `method` 다 (§6.2.10 과 §6.11.3). 한 뜻에 한 철자를 주기 위해서다 — 같은 글자가 네 가지를 뜻하면, 무엇을 읽고 있는지는 이름이 어디서 왔는지를 알아야만 정해진다.

> **쉬운 말로**
>
> 익숙한 표기(`1 + 2`)와 달라 보이지만 규칙은 하나다 — **이름이 먼저, 인자가 뒤.** 중위 표기가 필요하면 `expr` 섬 안에서 쓸 수 있고, 그 규칙은 §6.3.4 에 있다.

6.1.6 개행은 닫지 아니한다 (Newline is not a closer)

1 개행 (newline) 은 폼을 닫지 아니한다. 사이띄개와 똑같이 다룬다.

2 폼은 자기 닫개에서 끝난다. 닫개는 점 `.` 하나다. 괄호는 닫개가 아니다(§6.1.5 (3a)).

2a `do … end` 는 **서로 짝인 괄호**다. `end` 는 자기 `do` 만 닫고, 블록 밖의 폼은 닫지 아니한다. 그러므로 블록 안의 문장은 저마다 자기 점으로 닫혀 있어야 하며, 닫히지 않은 채 `end` 를 만나면 번역이 거부된다(`E-DOT-MISSING`).

2b **블록을 가진 것도 제 점으로 닫힌다.** 블록을 몸으로 갖는 문장(`if` · `while` · 네 되풀이 · `match` · `case` · `region` · `borrow` · `task_group` · `pipe`)과 선언(`fn` · `proc` · `struct` · `enum` · `trait` · `actor` · `state` · `contract` · `test`)은 `end` 다음의 점에서 끝난다 — `if c. do … end .` · `fn f … do … end .` · `else` 로 이어지는 사슬은 한 문장이고, 점은 사슬의 맨 끝에 하나다(`if a. do … end else do … end .`). 그 점이 빠지면 번역이 거부된다(`E-DOT-MISSING`).

2c 블록을 품은 **값** — `lit t do … end .` · `pipe xs. do … end .` — 도 제 점으로 닫히고, 그 값을 쓰는 문장은 다시 자기 점으로 닫힌다(`let x lit t do … end . .`).

2d 블록은 그것을 여는 머리 없이 홀로 설 수 없다(`E-BLOCK-NOHEAD`).

2e **아무것도 닫지 않는 점은 없다.** 이미 닫힌 폼 뒤에 겹친 점(`if c. . do`), op 머리의 이름 바로 뒤의 점(`fn f . input …`), 머리 절 가운데의 점(`input a . u64 .`)은 적합하지 아니하다(`E-CLOSER-EXTRA`). 타입은 제 문법으로 끝나므로 점을 갖지 아니한다(`input b slice u8 .` 의 점은 절을 닫는다). 머리 절은 점 하나

로 닫히고, 점 뒤의 낱말은 다음 절의 낱말이거나 `do` 다. `asm` 절은 제 항목(`reg a .`)을 점으로 나눈다. 절 낱말(`vector` · `output` · `effects` …)은 입력의 이름이 될 수 없다(`E-NAME-CLAUSE`).

거부되는 예제 (rejected) — 아무것도 닫지 않는 점

소스 코드 (source)

```
module ex_extra_dot .

fn head input data slice u8 . . output u8 .
  requires ge len data. . 1 . .
do
  return idx data. 0 . .
end .
```

진단: `E-CLOSER-EXTRA`

도해 (diagram) — end 는 블록만 닫고, 마침표는 그 블록을 가진 것을 닫는다

```
if eq a. 0 . do return 1 . end .      ← 마지막 점이 if 문을 닫는다
                └──────────┘ |        ← └┘ 는 if 의 블록, | 는 if 의 마침표

let p lit pt do x 1 . end . .         ← 앞의 점이 lit 을, 뒤의 점이 let 을 닫는다
      └──────────────────┘ | |
```

거부되는 예제 (rejected) — 블록으로 끝난 문장도 제 점으로 닫는다

소스 코드 (source)

```
module ex_dot_after_end .

fn f input a u64 . output u64 . do
  if eq a. 0 . do return 1 . end
  return a. .
end .
```

진단: `E-DOT-MISSING`

거부되는 예제 (rejected) — 블록을 값으로 쓰는 문장은 자기 점으로 닫는다

소스 코드 (source)

```
module ex_dot_missing .

struct pt do
  x u64 .
end .

fn f output u64 . do
  let p lit pt do x 1 . end
  return field p. x . .
end .
```

진단: `E-DOT-MISSING`

거부되는 예제 (rejected) — 머리 없는 블록

소스 코드 (source)

```
module ex_block_nohead .

fn f input a u64 . output u64 . do
  do
    return 1 .
  end .
  return a. .
end .
```

진단: `E-BLOCK-NOHEAD`

쉬운 말로

왜 `end` 가 자기 `do` 만 닫는가. `end` 가 바깥 문장까지 닫으면, 한 `end` 가 무엇을 끝냈는지 알려면 그 블록을 **누가 품었는지**를 거슬러 올라가 봐야 한다. 그래서 규칙은 둘뿐이다: **`do` … `end` 는 괄호처럼 짝을 이루고, 이름이 연 것은 점이 닫는다.** 블록을 몸으로 갖는 문장도, 블록을 값으로 쓰는 문장도 같은 규칙으로 끝난다 (2026-10-09).

3 그러므로 줄을 어디서 나누어도 뜻이 같다. 줄을 이어 쓰는 표시를 두지 아니한다 — 행끝 이음표도, 줄 끝의 쉼표도, 들여쓰기 규칙도 없다.

4 세미콜론 `;` 도 닫개가 **아니다.** 닫개의 철자는 **하나**다.

예제 (example) — 긴 폼은 줄을 나눠도 한 폼이다

소스 코드 (source)

```
module ex_newline .

rem 개행은 공백이다. 폼은 자기 닫개 `.` 에서 끝난다.
fn poly input a u64 . input b u64 . output u64 . do
  return add mul a. 2 .
             mul b. 3 . . .
end .
```

실행 결과 (output)

```
poly(5, 4) = 22
```

쉬운 말로

줄을 어디서 나누든 뜻이 같다. 이어 쓰는 방법을 따로 배울 것이 없다 — C 의 행끝 `\\` 도, 줄 끝의 쉼표도, 들여쓰기 규칙도 없다.

참고 (informative)

개행이 닫개였던 때가 있었다. 2026-08-27 이전에는 개행이 폼을 닫았다. 그래서 줄을 이어 쓰는 방법을 셋 배워야 했고, 잘못 나누면 뜻이 조용히 바뀌었다. 그 규칙을 없앴다. 닫개는 오직 `.` 이고, 개행은 공백이다.

거부되는 예제 (rejected) — 세미콜론은 닫개가 아니다

소스 코드 (source)

```
module ex_semi .

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

진단: `E-VOCAB-REMOVED`

> **쉬운 말로**
>
> 세미콜론은 한때 닫개의 **세 번째 철자**였다 — 처리기가 그것을 만나면 글자 그대로 점을 냈다. 같은 것을 가리키는 이름이 셋이면 읽는 사람이 셋을 다 알아야 하므로, 순수한 동의어는 없앴다(§1.3).

> **쉬운 말로**
>
> 규칙이 이러해도, 실제로 쓰는 코드는 **점을 꼬박꼬박 적는 편**이다. 개행이 닫는지 아닌지를 줄마다 따지는 것보다 늘 적는 쪽이 읽기 쉽기 때문이다. 개행 닫개는 "점을 안 적어도 된다"는 허락이라기보다, **줄이 곧 문장이라는 사실**을 문법이 인정하는 것에 가깝다.

6.1.7 닫히지 않은 것

제약

1 연 것은 닫아야 한다. 문자열 리터럴과 문자 리터럴은 **같은 줄에서** 닫는 따옴표를 만나야 하고, 텍스트 리터럴과 `note` 뭉치는 닫는 줄을 만나야 하며(§6.1.4 (4) · §6.1.1 (1)), 블록은 `end` 를, 묶음은 `)` 를 만나야 한다.

진단

표 16 — 닫히지 않은 것과 그 진단

진단	무엇이 안 닫혔는가
`E-STR-UNTERM`	글월이 파일 끝까지 닫히지 아니하였다
`E-STR-NEWLINE`	글월 안에 **줄바꿈**이 들어왔다 — 글월은 한 줄이다(여러 줄은 텍스트 리터럴이다)
`E-CHAR-UNTERM` · `E-CHAR-NEWLINE`	낱글자가 닫히지 아니하였다
`E-NOTE-UNTERM`	적바림 뭉치가 닫히지 아니하였다
`E-NOTE-TERM`	`note` 뒤에 태그가 없다
`E-TEXTLIT-UNTERM` · `E-TEXTLIT-TERM`	텍스트 리터럴의 맺음말이 없거나 어긋났다
`E-TEXTLIT-TAIL` · `E-NOTE-TAIL`	텍스트 리터럴이나 적바림 뭉치를 닫는 태그 뒤에 다른 글자가 왔다 — 닫는 줄에는 태그만 선다
`E-TEXTLIT-OPENER`	텍스트 리터럴을 여는 줄에서 태그 뒤에 처리기가 아닌 것이 왔다
`E-BLOCK-UNCLOSED`	블록이 닫히지 아니하였다
`E-GROUP-UNCLOSED`	묶음이 닫히지 아니하였다

참고

> **참고 (informative)**
>
> 이 진단들은 대개 **소스의 끝에서** 난다. 여는 표시는 그것이 닫히기 전에는 무엇이 잘못인지 알 수 없기 때문이다.

6.1.8 소스가 갖추어야 하는 것

제약

1 소스는 **온전한 UTF-8** 이어야 한다.

2 소스의 글자는 모두 어느 `token` 이나 `separator` 에 속해야 한다(§6.1). 어디에도 속하지 않는 글자가 올 수 없다.

3 정수 리터럴은 처리기가 번역할 때 담을 수 있는 크기여야 한다.

4 폼과 블록을 겹쳐 쓰는 깊이는 처리기가 따라갈 수 있는 한도 안이어야 한다. 처리기는 한도를 넘은 프로그램을 조용히 잘라 다른 프로그램으로 번역하지 아니한다(§4.5).

진단

표 17 — 소스의 진단

어긴 것	진단
(1) — UTF-8 이 아닌 바이트가 있다	`E-LEX-UTF8`
(2) — 어느 토큰에도 속하지 않는 글자가 있다	`E-CHAR`
(3) — 정수 리터럴이 너무 크다	`E-LIT-RANGE`
(4) — 너무 깊이 겹쳐 썼다	`E-NEST-DEPTH`

예제

> **거부되는 예제 (rejected) — 어느 낱말에도 속하지 않는 글자**
>
> **소스 코드 (source)**
>
> ```
> module ex_stray_char .
>
> fn f output u8 . do return 1 @ . end .
> ```
>
> 진단: `E-CHAR`

참고

> **주의 —**
>
> `..` 는 이 언어에 없다(`E-DOT-DOUBLE`). 한때 있던 철자이며, **버린 철자를 새 뜻으로 되살리지 아니한다** — 옛 글을 읽는 사람이 옛 뜻으로 읽기 때문이다.

> **참고 (informative)**
>
> 수 리터럴의 모양에 대한 제약(밑줄의 자리 · 진법 표시 뒤의 자릿수 · 수 뒤에 붙은 글자)은 §6.1.4 (5) (7)에 있다. `0x` 만 적으면 0 이 아니라 거부된다 — 0 은 `0x0` 이라 적는다. 수의 접미사(`3u8`)도 팔진 표기(`0o7`)도 없다.

6.2 타입 (Types)

1 모든 값은 정확히 하나의 타입 (type) 을 가진다. 타입은 그 값이 가질 수 있는 것과, 그 값에 할 수 있는 일을 함께 정한다.

2 로우엔트의 모든 타입은 **크기와 표현이 정해져 있다.** 처리기가 기계에 따라 마음대로 정하는 자리가 없다.

2a 처리기는 식마다 그 값의 타입을 정한다. 타입이 적힌 자리 — 묶는 이름 · 대입 · op 의 인자와 반환 · 줄의 원소 · 묶음의 칸 — 에 처리기가 타입을 정할 수 없는 값을 두면 번역이 거부된다(`E-TYPE-UNKNOWN`). 타입을 정하지 못한 값은 검사할 수 없으므로, 그 자리를 «맞다» 로 지나가게 두지 아니한다.

> **거부되는 예제 (rejected) — 값을 내지 않는 낱말을 수로 묶는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_type_unknown .
>
> fn f input s mut slice u64 . output u64 . do
>   let x u64 swap s. 0 1 . .
> ```

```
  return x. .
end .
```

진단: `E-TYPE-UNKNOWN`

> **쉬운 말로**
>
> C 를 아는 사람에게: C 의 `int` 는 기계마다 크기가 다를 수 있다. 로우엔트에는 그런 타입이 없다 — `u32` 는 어디서나 32 비트다. 그래서 "이 기계에서 몇 바이트지?"를 물을 일이 없다.

6.2.1 타입의 갈래

1 타입은 다음 갈래로 나뉜다.

표 18 — 타입의 갈래

갈래	무엇인가
정수	`u8 u16 u32 u64 usize i8 i16 i32 i64 idiff`
부동소수점	`f32 f64`
참거짓	`bool`
묶음	`struct`(이름 붙은 칸의 모음) · `enum`(여럿 중 하나)
줄	`array n t`(길이가 고정된 줄) · `slice t`(길이를 함께 갖는 구간) · `matrix k d… t`(크기가 고정된 여러 차원의 줄)
답	`result t e`(성공 값 또는 오류) · `option t`(값이 있거나 없음)
가리킴	`ref t`(읽기 참조) · `mut_ref t`(쓰기 참조) · `owned t`(소유)
권한	`cap k`(효과를 낼 자격, §7.2)
흐름	`task t`(흐름의 핸들 — 그 흐름이 낼 값의 타입이 `t`, §10.3) · `chan t`(채널 — 담는 값의 타입이 `t`, §10.6)

> **참고 (informative)**
>
> 이 목록에 **포인터가 없다**. `ref`·`slice`·`owned` 가 포인터가 하던 일을 갈라 맡는다 — "무엇을 가리키는가"와 "얼마나 있는가"와 "누가 없앨 책임이 있는가"는 서로 다른 질문이고, 하나의 포인터가 셋을 다 답하려 하면 어느 것도 검사할 수 없게 된다.

6.2.2 정수 타입

1 정수 타입의 이름은 부호와 폭으로 이루어진다. `u` 는 부호 없음, `i` 는 부호 있음이며, 뒤의 숫자가 비트 폭이다.

2 정수는 2 의 보수로 표현된다(§5.3).

3 `usize` 와 `idiff` 는 주소 공간의 **크기와 첨자**를 셀 수 있을 만큼 넓은 정수다. 이 둘은 `u64`·`i64` 와 폭이 같더라도 **다른 타입**이며, 서로 자동으로 바뀌지 아니한다. `usize` 는 길이와 첨자를, `idiff` 는 **두 위치의 차이**를 적는 타입이다.

3-1 `usize` 와 `idiff` 의 폭은 같고, **구현이 정한다**(구현 정의). 그 폭은 16 비트 이상이며, 그 구현이 갖는 가장 넓은 정수 타입의 폭을 넘지 아니한다. 구현은 그 폭을 문서에 적는다. `usize` 는 부호가 없고 `idiff` 는 2 의 보수의 부호 있는 정수이므로, `idiff` 가 담는 양수는 `usize` 가 담는 범위의 절반이다.

3-2 `usize` · `idiff` 는 **다른 어떤 정수 타입과도 섞이지 아니한다.** 넓히기의 표(§6.2.5 (4))는 이 둘을 포함하지 아니하며 — `u8` 에서 `usize` 로도, `usize` 에서 `u64` 로도 자동 변환이 없다. 산술과 비교의 두 피연산

자 가운데 하나만 `usize`(또는 `idiff`)이면 번역이 거부된다(`E-TYPE-NOMINAL`). `usize` 와 `idiff` 사이도 같다. 수 리터럴은 자리의 타입을 따르므로 이 규칙에 걸리지 아니한다.

3-3 `usize` · `idiff` 와 다른 정수 타입 사이의 변환은 **`cast` 로 적는다**(§6.2.16 (2)). `cast` 는 값이 목표 타입에 들어가면 그 값을 내고, 들어가지 않으면 트랩한다. `widen` 과 `narrow` 는 원천이나 목표가 `usize` · `idiff` 이면 번역이 거부된다 (`E-TYPE-NOMINAL`) — 그 변환이 넓히는지 좁히는지가 구현의 폭에 달려 있어, 방향을 말하는 낱말로는 어느 구현에서나 참인 글을 적을 수 없기 때문이다. 들어가지 않을 때의 처분을 이름이 고르는 `narrow_try` · `narrow_sat` · `narrow_wrap` 은 `usize` · `idiff` 를 목표로 받는다.

3-4 리터럴이 그 구현의 `usize`(또는 `idiff`)에 들어가지 않으면 번역이 거부된다(`E-TYPE-WIDTH`). 이것은 **정수 변환에서** 같은 글의 판정이 구현에 따라 달라지는 유일한 자리다 — 그 값은 그 구현의 주소 공간에 없는 크기이므로, 실행 중의 트랩보다 이르게 거절한다. 대상이 갖지 못한 것을 쓴 글의 거절(§5.6 (4) · (4-2) — 힙 · 부동소수점 · 128 비트 정수 · 원자 폭)은 이 문장 밖이다.

> **주의 — `usize` 의 폭에 기대지 않는 글**
>
> 같은 글이 16 · 32 · 64 비트 구현 어디서나 서려면 — ① 길이 · 첨자 · 바이트 수만 `usize` 로 적고, 파일 안의 위치 · 시각 · 해시 · 프로토콜의 칸은 폭이 정해진 타입으로 둔다. ② 둘이 만나는 셈은 **넓은 쪽에서** 한다 (`add off. cast u64 len b. . . .`). ③ 바깥에서 온 수를 `usize` 로 옮길 때는 `narrow_try` 로 받아, 들어가지 않는 것을 프로그램이 거절한다. ④ `usize` 의 셈은 그 폭에서 넘치므로(§6.3), 크기를 곱하는 자리는 `chk_mul` 로 받는다. ⑤ 구조체의 `usize` 칸은 구현마다 크기가 다르므로, 바깥으로 나가는 배치에는 두지 아니한다. ⑥ 두 위치의 차이를 `idiff` 로 받을 때 `cast idiff` 는 `usize` 범위의 위쪽 절반에서 트랩한다.

> **예제 (example) — `usize` 와 다른 정수 사이는 `cast`**
>
> **소스 코드 (source)**
>
> ```
> module ex_usize .
>
> fn bucket input xs slice u8 . input h u64 . output u8 .
>   requires gt len xs. . 0 . .
> do
>   let slot u64 mod h. cast u64 len xs. . . . .
>   return idx xs. cast usize slot. . . .
> end .
>
> fn checked input want u64 . output usize . do
>   let p option usize narrow_try usize want. . .
>   guard is_some p. . else return 0 . .
>   return some_value p. . .
> end .
> ```

> **거부되는 예제 (rejected) — `usize` 와 `u64` 를 그대로 섞었다**
>
> **소스 코드 (source)**
>
> ```
> module ex_usize_mix .
>
> fn end_of input start usize . input more u64 . output u64 . do
>   return add start. more. . .
> end .
> ```
>
> 진단: `E-TYPE-NOMINAL`

> **거부되는 예제 (rejected) — `usize` 를 `widen` 으로 옮겼다**
>
> **소스 코드 (source)**

```
module ex_usize_widen .

fn as_u64 input n usize . output u64 . do
  return widen u64 n. . .
end .
```

진단: `E-TYPE-NOMINAL`

3b u128 · i128 은 폭 128 의 정수다. 이 둘은 **128 비트 정수를 가진 구현의 대상에서만** 선다 — 그렇지 않은 대상에서 이 낱말을 쓰면 번역이 거부된다(`E-INT128-TARGET`). 구현은 이 둘이 서는 대상을 문서에 적는다.

3b-1 다른 폭의 정수에서 128 비트 자리로는 **자동 넓히기가 없다** — `widen u128 …` · `widen i128 …` 으로 적는다(`E-TYPE-WIDTH`). 산술 · 비교 · 비트 연산의 두 피연산자는 둘 다 128 비트이거나 둘 다 아니다. 수 리터럴은 자리의 타입을 따른다 — 64 비트를 넘는 리터럴은 128 비트 자리에만 들어가고, 다른 자리에 오면 번역이 거부된다(`E-LIT-RANGE`, §6.1).

3b-2 128 비트 정수는 스칼라 자리(지역 · 입력 · 결과 · `option` · `result` 의 알맹이)에 더해 구조체의 칸과 슬라이스 · 배열 · 뷰 · 스택의 원소에 선다. 그 바이트 배치는 대상의 C 컴파일러가 128 비트 정수에 주는 것과 같다 — 크기 16 바이트, 정렬 16 바이트, 작은 끝이 먼저(`big` 을 적은 칸은 뒤집는다). 벡터(`vec` · `matrix`)의 원소와 `extern` 의 경계에 두면 번역이 거부된다 (`E-INT128-PLACE`) — 128 비트 정수 레인을 가진 벡터 장치가 없고, C 경계의 ABI 는 아직 열지 아니하였다.

3b-3 명령줄로 받는 128 비트 입력은 적은 십진수 그대로다. 그 타입의 범위를 벗어나는 수(`u128` 의 음수 · 2^{128} 이상, `i128` 의 $\pm 2^{127}$ 밖)는 잘라 넣지 않고 거부된다 — 진입 경계의 타입 범위 검사와 같다.

예제 (example) — 64 비트 두 수의 곱 전체를 128 비트로

소스 코드 (source)

```
module ex_u128 .

fn wide_mul input a u64 . input b u64 . output u128 . do
  let x u128 widen u128 a. . .
  let y u128 widen u128 b. . .
  return mul x. y. . .
end .

fn top_half input a u64 . input b u64 . output u64 . do
  return cast u64 shr wide_mul a. b. . 64 . . .
end .
```

예제 (example) — 구조체의 칸에 128 비트 정수 — 배치는 C 와 같다

소스 코드 (source)

```
module ex_u128_field .

struct acc do
  lo u64 .
  big u128 .
end .

fn acc_bytes output usize . do
  return size_of acc . .
end .
```

거부되는 예제 (rejected) — C 경계에 128 비트 정수를 두었다

소스 코드 (source)

```
module ex_u128_extern .

export extern fn take input a u128 . output u64 . do
  return 0 .
end .
```

진단: `E-INT128-PLACE`

3a `usize` 는 **포인터를 담는 타입이 아니다.** 그 폭은 주소의 폭을 따르지만, 포인터 값은 주소보다 클 수 있고(§5.3 (4a)) 정수에서 되살릴 수 없다. 크기 · 길이 · 첨자에는 `usize` 를 쓰고, 가리키는 것은 슬라이스와 참조로 적는다.

표 19 — 정수 타입의 범위

타입	폭	범위
`u8`	8	0 … 255
`u16`	16	0 … 65 535
`u32`	32	0 … 4 294 967 295
`u64`	64	0 … $2^{64}-1$
`i8`	8	−128 … 127
`i16`	16	−32 768 … 32 767
`i32`	32	−2 147 483 648 … 2 147 483 647
`i64`	64	-2^{63} … $2^{63}-1$
`u128`	128 (대상에 따라 — (3b))	0 … $2^{128}-1$
`i128`	128 (대상에 따라 — (3b))	-2^{127} … $2^{127}-1$
`usize`/`idiff`	16 이상(구현 정의)	폭에 따른다 — (3-1)

6.2.3 참거짓 타입

1 `bool` 은 참과 거짓 두 값만 갖는다. 표현은 1 바이트이며, 유효한 값은 0 과 1 뿐이다.

2 `bool` 과 정수는 **서로 바뀌지 아니한다.** 정수를 조건 자리에 쓸 수 없고, `bool` 을 숫자로 셈할 수 없다.

거부되는 예제 (rejected) — 정수를 조건 자리에 쓸 수 없다

소스 코드 (source)

```
module ex_cond .

proc p input n u32 . output u8 . effects none .
do
  if n. do return 1 . end .     rem `n` 은 bool 이 아니다
  return 0 .
end .
```

진단: `E-TYPE-COND`

> **주의 — 0 은 거짓이 아니다**
>
> C 계열 언어는 0 을 거짓으로, 그 밖을 참으로 다룬다. 로우엔트는 그러지 아니한다. `if n. do` 처럼 정수를 조건에 쓰면 번역이 거부된다 — `if gt n. 0 . do` 처럼 **무엇을 묻는지** 적어야 한다. 묻는 바가 소스에 적히지 않으면, 읽는 사람이 그것을 짐작해야 한다.

6.2.4 부동소수점 타입

1 `f32` 와 `f64` 는 각각 IEEE 754 의 이진 32 비트·64 비트 형식이다.

2 부동소수점 연산은 **피연산자의 폭에서** 일어난다. `f32` 끼리의 연산은 `f32` 로 계산하며, 처리기가 몰래 더 넓은 정밀도로 계산하지 아니한다.

2a 셈 하나는 **반올림 한 번**이다. 처리기는 두 셈을 반올림 한 번으로 합치지 아니한다 — 곱하고 더하기(`add mul a. b. . c.`)를 곱셈 · 덧셈 하나의 명령(FMA)으로 바꾸지 아니한다. 반올림은 «가장 가까운 값, 비기면 짝수» 하나이고, 프로그램이 바꿀 수 없다. C 로 낸 코드도 이 약속을 소스에 담는다 — 그 코드를 짓는 C 컴파일러가 스스로 합치지 않도록 지시를 적는다.

3 정수와 부동소수점 수는 서로 자동으로 바뀌지 아니한다. 바꾸려면 명시적으로 적어야 한다.

4 셈의 결과가 NaN 이면 그것이 NaN 이라는 것은 정해져 있다. 그 NaN 의 부호 비트와 나머지 비트는 **자유 선택 (free choice)** 이다(FC-0004).

> **자유 선택 (free choice) — FC-0004 — 셈이 낸 NaN 의 부호**
>
> 부동소수점 셈(레인마다의 셈과 `reduce_*` 를 포함한다)의 결과가 NaN 일 때, 그 부호 비트와 나머지 비트는 처리기가 고른다. 결과가 NaN 이라는 것과, NaN 이 아닌 결과의 값은 어느 쪽에서도 같다. 차이가 드러나는 자리는 그 값을 찍을 때(`nan` 과 `-nan`)와 비트로 읽을 때다. IEEE 754 도 이 자리를 정하지 않고, 기계와 C 컴파일러가 NaN 둘이 만나는 자리나 `0.0 / 0.0` 에서 서로 다른 부호를 낸다. 한쪽으로 못 박으면 셈마다 고치는 명령이 붙는다. NaN 인지 알고 싶으면 `ne x. x.` 처럼 그 값 자신과 견준다 — 부호를 묻지 않는다.

6.2.5 타입 사이의 변환

1 로우엔트에는 **값을 잃는 암묵적 변환이 없다.** 좁아지는 자리는 언제나 소스에 적혀 있다. 처리기가 스스로 하는 변환은 아래의 넓히기뿐이고, 그 밖의 변환을 적는 낱말은 `cast` 다(§6.2.16).

2 값을 잃지 않는 변환을 넓히기 (widening) 라 한다. 넓히기는 값을 바꾸지 않으므로 처리기가 자동으로 할 수 있다 — `u8` 값을 `u32` 를 받는 자리에 그대로 넘길 수 있다.

2a 다만 **넓히기를 소스에 적을 수도 있다.** `widen u64 x. .` 는 `x` 를 `u64` 로 넓힌다. 결과가 같더라도, 폭이 바뀌는 자리를 눈에 보이게 하고 싶을 때 쓴다.

2b 128 비트 정수(`u128` · `i128`)로의 넓히기는 **자동이 아니다** — 반드시 `widen` 으로 적는다(§6.2.2 (3b-1)).

> **쉬운 말로**
>
> 정리하면 이렇다 — **넓히는 쪽은 자동, 좁히는 쪽은 손으로.** 넓히기는 값이 그대로라 실수할 여지가 없지만, 좁히기는 값이 사라질 수 있어서 저자가 그것을 알고 있음을 소스에 남겨야 한다.

3 값을 잃을 수 있는 변환을 좁히기 (narrowing) 라 하며 `narrow` 로 적는다. 좁히기는 값이 목표 타입에 들어가지 않으면 **트랩한다**. 조용히 잘리지 아니한다.

4 넓히기가 허용되는 관계는 다음과 같다. 여기 없는 조합은 넓히기가 아니며, 적으면 번역이 거부된다(`E-WIDEN-KIND` · `E-WIDEN-SIGN` · `E-WIDEN-NARROW`).

표 20 — 넓히기가 허용되는 관계

관계	설명
`uN` ⟶ `uM` (N ≤ M)	부호 없는 정수를 더 넓은 부호 없는 정수로
`iN` ⟶ `iM` (N ≤ M)	부호 있는 정수를 더 넓은 부호 있는 정수로

`uN` ⟶ `iM` (N < M)	부호 없는 정수를 **더 넓은** 부호 있는 정수로
`iN` ⟶ `uM`	**허용되지 아니한다** — 음수가 표현되지 않는다
`f32` ⟶ `f64`	부동소수점 수를 더 넓은 부동소수점으로
정수 ↔ 부동소수점	**넓히기가 아니다** — 갈래가 다르면 명시적으로 바꿔야 한다
`usize` · `idiff` ↔ 다른 정수	**넓히기가 아니다** — 폭이 구현에 달려 있다. `cast` 로 적는다(⟦§6.2⟧ (3-3))

도해 (diagram) — 넓히기 — 화살을 따라가면 값이 그대로 들어간다

```
u8 ⟶ u16 ⟶ u32 ⟶ u64          f32 ⟶ f64
  \      \      \
   ▼      ▼      ▼
i8 ⟶ i16 ⟶ i32 ⟶ i64
같은 폭으로 내려가는 화살(u8 → i8)도, i 줄에서 u 줄로 가는 화살도 없다
usize · idiff 는 이 그림에 없다 — 어느 쪽으로도 화살이 없고, 건너는 낱말은 cast 다
```

예제 (example) — 넓히기와 좁히기

소스 코드 (source)

```
let small u8 200 .
let wide u64 widen u64 small. . .
let back u8 narrow u8 wide. . .
```

주의 — 음수를 부호 없는 타입으로 넓힐 수 없다

`i8` 의 −1 을 `u64` 로 넓히면 값이 아주 큰 수로 뒤바뀐다. 그것은 넓히기가 아니라 **다른 값이 되는 일**이므로 이 언어는 넓히기로 인정하지 아니한다.

6.2.6 줄 — 배열과 슬라이스

1 `array n t` 은 타입 `t` 의 값이 정확히 `n` 개 놓인 줄이다. 길이는 타입의 일부이며 번역 시점에 정해진다.

1a **길이 `n` 을 먼저 정수 리터럴로, 원소 타입을 뒤에** 적는다 — `vec n t` · `matrix k d… t` 와 같은 차례다. 원소 타입이 맨 뒤에 오므로 타입이 제 끝을 스스로 안다. 반대로 적으면(`array t n` — 길이 자리에 타입) 거부된다(`E-TYPE-ORDER`. `lowentc --fmt` 가 새 차례로 옮긴다). 길이가 리터럴이 아니면 거부된다(`E-TYPE-ARRAY`).

1b op 의 입력 `input x array n t .` 은 길이가 정확히 `n` 인 `slice t` 를 받는다는 뜻이며, 그 길이는 `requires eq len x. . n. . .` 과 같이 **진입에서 검사된다**.

1c op 은 배열을 **돌려줄 수 있다** — `output array n t .` 또는 이름을 붙여 `output r array n t .`(§6.4.1 (3e)). 배열에는 값으로 돌려주는 길이 없으므로 **부르는 쪽이 자리를 준다**:

- 부르는 쪽은 결과를 이름에 묶는다 — `var x array n t f a … .`(`let` 도 같다). 그 이름의 틀 안 바이트가 결과의 자리이고,

부름 받는 쪽은 거기에 바로 짓는다(베끼지 않는다).

- 같은 배열을 돌려주는 op 안에서 `return g … .` 은 자기 자리를 그대로 넘긴다. 다만 `g` 에 건네는 인자가 그 자리와 같은 바이트를

볼 수 있으면(이름 붙은 결과 자신이거나, 그것에서 나온 이름 — 초기식이나 `set` 의 오른쪽이 그것을 언급한 지역) 처리기는 **임시 자리에 받아 한 번 베낀다** — `g` 가 쓰는 동안 읽는 쪽이 새 값을 보지 않으므로 답은 겹치지 않을 때와 같다. 겹치지 않는다고 보일 때(인자가 그런 이름을 하나도 언급하지 않을 때)만

바로 넘긴다. 블록 밖의 return(guard … else return …)은 문장을 끼울 자리가 없어 이 규칙이 서지 않는다 — 아래의 E-RESULT-PLACE 가 그대로다.

- 그 밖의 자리(식의 한가운데 · 다른 부름의 인자 · copy 의 원천)에서 배열을 돌려주는 op 을 부르면 자리가 없으므로

번역이 거부된다(E-RESULT-PLACE). 먼저 이름에 묶는다.

- 이름 붙은 결과는 0 에서 시작한다(칸마다 0). 이름 없는 결과의 return v. . 는 v 를 자리에 베껴 돌려준다. 이것은 블록 안의

return 이어야 한다 — guard … else return v. . . 처럼 블록 밖이면 거부된다(E-RESULT-PLACE). 그런 op 은 결과에 이름을 붙인다.

- 원소는 수 타입이고 길이는 정수 리터럴이다(그 밖은 E-TYPE-ARRAY).

1d **배열 안의 배열** — 원소 타입이 다시 배열이어도 된다. array n array m t 는 길이 m 인 줄 n 개가 줄 우선으로 빈틈 없이 놓인 줄이다(C 의 t a[n][m] 과 같은 배치). 원소 타입이 맨 뒤에 오므로 괄호가 들지 않는다((1a)).

- idx a. i. . 는 줄 i 의 바이트를 **보는** slice t(길이 m, 복사 없음)다 — (9a) 의 줄의 나열과 같은 규칙을 따른다. 바깥이

mut 이면 그 줄의 칸도 쓸 수 있고, 한 줄을 통째로 바꾸는 set idx a. i. . <줄> . 은 없다(E-TYPE-SET).

- op 의 입력 input x array n array m t . 은 줄의 슬라이스 slice array m t 를 받으며 줄 수가 n 인지 **진입에서 검사된다**((1b)).

줄의 나열도, 바이트 위의 줄 보기(§8.7 (7a))도 그대로 넘길 수 있다.

- 구조체의 칸이면 원소 n × m 개의 바이트를 **레코드 안에** 가진다(§6.2.7 (1a) 의 배열 칸과 같다 — 구조체를 베끼면 함께 베껴진다).
- 값은 lit array n array m t <줄>… [_] . 로 짓는다. 줄마다 lit array m t … 이고, 끝의 _ 는 남은 줄을 0 으로 채운다((6a)).
- 원소는 크기 있는 수와 bool 이다. 겹의 수에는 한도가 없다 — array 2 array 3 array 4 u8 은 길이 4 인 줄 3 개짜리 판 2 개(C 의

u8 a[2][3][4])이고, idx a. i. . 는 판 i 를 **보는** 줄의 슬라이스 slice array 4 u8 이다. 겹마다 idx 가 한 겹씩 벗긴다. 줄의 나열 · 줄 보기의 줄도 배열 안의 배열일 수 있다(slice array 3 array 4 u8 · view_array array 3 array 4 u8 b.).

- 겹의 깊이는 타입의 일부다 — 깊이가 다른 줄의 슬라이스를 넘기거나 묶으면 거부된다(E-TYPE-ARG · E-TYPE-LET). 줄의 슬라이스는

원소의 슬라이스 자리에도 들지 않는다(idx 가 수 대신 줄을 낸다).

- op 은 배열 안의 배열도 **돌려줄 수 있다** — output r array n array m t .. (1c) 와 같이 부르는 쪽이 자리를 주고(var x array n array m t f … .),

이름 붙은 결과는 0 인 줄들로 시작한다.

> **예제 (example) — 배열 안의 배열**

소스 코드 (source)

```
module ex_nested_array .

fn rowsum input g array 2 array 3 u8 . input r usize . output u64 .
  requires lt r. 2 . .
do
  var s u64 0 .
  for x idx g. r. . do
```

```
    set s. add s. widen u64 x. . . .
  end .
  return s. .
end .

export fn table output u64 . do
  var g lit array 2 array 3 u8 lit array 3 u8 1 2 3 . _ . .
  set idx idx g. 1 . 2 . 7 .
  return add mul rowsum g. 0 . 100 . rowsum g. 1 . . .
end .
```

예제 (example) — 세 겹 — 겹마다 idx 가 아래 겹을 본다

소스 코드 (source)

```
module ex_nested_three .

export fn f output usize . do
  var g lit array 2 array 3 array 4 u8 _ . .
  set idx idx idx g. 1 . 2 . 3 . 9 .
  let plane slice array 4 u8 idx g. 1 . .
  return add len plane. . cast usize idx idx plane. 2 . 3 . . . .
end .
```

거부되는 예제 (rejected) — 깊이가 다른 줄의 슬라이스

소스 코드 (source)

```
module ex_nested_depth .

fn rows input g array 3 array 4 u8 . output usize . do
  return len g. . .
end .

export fn f output usize . do
  let c lit array 2 array 3 array 4 u8 _ . .
  return rows c. . .
end .
```

진단: `E-TYPE-ARG`

예제 (example) — 배열을 돌려주는 op — 부르는 쪽이 자리를 준다

소스 코드 (source)

```
module ex_array_result .

fn squares input n u64 . output r array 3 u64 .
  requires le n. 1000 . .
do
  repeat i usize 3 do
    let v u64 add n. cast u64 i. . . .
    set idx r. i. . mul v. v. . .
  end .
  return r. .
end .
```

```
export fn total input n u64 . output u64 .
  requires le n. 1000 . .
do
  let s array 3 u64 squares n. . .
  return add add idx s. 0 . idx s. 1 . . idx s. 2 . . .
end .
```

거부되는 예제 (rejected) — 배열을 돌려주는 op 을 식 한가운데서 부른다 — 결과의 자리가 없다

소스 코드 (source)

```
module ex_array_result_expr .

fn squares input n u64 . output r array 3 u64 .
  requires le n. 1000 . .
do
  return r. .
end .

export fn first input n u64 . output u64 .
  requires le n. 1000 . .
do
  return idx squares n. . 0 . .
end .
```

진단: E-RESULT-PLACE

2 slice t 는 타입 t 의 값이 연속으로 놓인 구간을 가리키는 것이며, **시작과 길이를 함께 갖는다.**

3 슬라이스의 길이는 len 으로 읽는다. len s. . 는 s 의 원소 개수이고, 그 타입은 usize 다(§6.2 (3a)). bitset_count · stack_count 와 capacity 가 내는 수도 usize 다. 권한 cap args 에 쓴 len 은 받은 인자의 수다.

3′ 첨자와 경계(idx 의 i, subslice 의 시작과 끝, 원자 연산의 첨자, alloc_bytes 의 capacity)는 usize **다**. 다른 정수 타입의 값을 그 자리에 대면 번역이 거부된다(E-TYPE-NOMINAL). 정수 리터럴은 그대로 적는다. 길이가 아닌 수와 만날 때는 변환을 적는다 — cast u64 n. . 는 usize 를 u64 로, cast usize x. . 는 u64 를 usize 로 옮기며, 어느 쪽이든 들어가지 않으면 트랩한다(§6.2 (3-3)).

3″ 원소에 쓰는 값은 그 슬라이스의 **원소 타입**이어야 한다 — set idx s. i. . v . 에서 v 가 원소 타입으로 넓혀지지 않으면 번역이 거부된다(E-TYPE-WIDTH · E-TYPE-SIGN · E-TYPE-NOMINAL). slice usize 와 slice u64 는 원소의 폭이 같은 구현에서도 **다른 타입**이다(E-TYPE-NOMINAL) — 위치만 담는 표는 slice usize 로 선언하고, 여러 뜻의 수를 함께 담는 표는 고정 폭으로 두고 읽고 쓰는 자리에 cast 를 적는다.

3‴ usize 는 **조각 하나의 것**을 센다: 그 길이 · 그 안의 첨자 · 한 번의 연산이 그 조각에서 읽거나 쓰거나 처리한 수. 그런 수는 그 조각의 길이를 넘지 못하므로 어느 구현에서도 usize 에 들어간다. 여러 번에 걸쳐 쌓이는 **누계**와 기억 장소 밖의 크기 · 위치 (파일의 크기와 위치 · 흐름으로 지나간 바이트의 합 · 메시지 전체의 비트 길이)는 주소 폭과 무관하게 클 수 있으므로 usize 가 **아니다** — 고정 폭 정수(u64)로 둔다. 해시 값 · 검사값 · 난수는 길이가 아니다: 그 값의 폭으로 둔다.

4 idx s. i. . 는 s 의 i 번째 원소다. 첫 원소의 번호는 0 이다.

5 i 가 len s. . 보다 작지 않으면 **트랩한다**. 처리기가 그 조건이 언제나 참임을 증명하면 그 검사는 사라진다(§6.4.6).

도해 (diagram) — 슬라이스는 시작과 길이를 함께 든다

```
memory   [ 10 ][ 20 ][ 30 ][ 40 ][ 50 ]
                ^
s = { start ────┘ , len 3 }
idx s 0 = 20 · idx s 2 = 40 · idx s 3 → trap (len s = 3)
```

예제 (example) — 슬라이스

소스 코드 (source)

```
module ex_slice .

rem 슬라이스는 시작과 길이를 함께 갖는다.
export fn head input data slice u8 . output u8 .
  requires ge len data. . 1 . .
do
  return idx data. 0 . .
end .
```

거부되는 예제 (rejected) — 원소 타입을 길이 앞에 적었다

소스 코드 (source)

```
module ex_array_order .

export fn last input xs array u64 4 . output u64 .
do
  return idx xs. 3 . .
end .
```

진단: E-TYPE-ORDER

6 **원소 나열 리터럴** — `lit array n t v₁ … vₖ .` 은 `array n t` 값, `lit slice t v₁ … vₖ .` 은 원소 k 개의 `slice t` 값이다. 나열은 **제 마침표로 닫히는 폼**이다: 문장의 끝에 오면 나열의 점과 문장의 점이 잇달아 오고(`let t lit array 4 u8 1 2 3 4 . .`), 한가운데 오면 점 하나로 닫고 다음 피연산자가 이어진다. `lit` 가 값의 타입을 보이므로 묶는 문장은 타입을 따로 적지 않는다.

6a 배열의 원소 수는 n 과 같아야 한다. 모자라면 나열의 **끝**에 _ 를 적어 «나머지 칸은 0» 이라고 말해야 하며, 넘치거나 _ 가 끝이 아닌 자리에 있거나 슬라이스 나열에 _ 가 있으면 거부된다(E-LIT-COUNT). 칸은 말없이 채워지지 않는다.

6b 각 원소는 t 에 들어가야 한다 — 부호 없는 t 에 음수도 들어가지 않는다(E-TYPE-WIDTH). `bool` 원소는 `true`·`false` 만이다.

6c **칸 골라 채우기** — `lit array n t do <번호> <값> . … [_ <값> .] end .` 은 번호를 적은 칸에 그 값을 넣는다. 번호는 0 부터 `n - 1` 까지의 정수 리터럴이고 한 번씩만 적는다. `_ <값> .` 은 끝에만 오며 «이름을 적지 않은 칸 모두» 를 뜻한다. 모든 칸이 정해져야 한다 — 남는 칸이 있는데 `_ <값> .` 이 없거나, _ 가 채울 칸이 없으면 거부된다 (E-LIT-COUNT). 번호가 상수가 아니거나 길이 밖이거나 두 번 나오거나 _ 가 끝이 아닌 자리에 있으면 거부된다 (E-LIT-INDEX). 값은 식이어도 되며 적은 차례로 계산되고, _ 의 값은 **한 번** 계산된다. 블록으로 적은 나열도 폼이라 제 점으로 닫힌다 — `let t lit array 8 u8 do 2 5 . _ 0 . end . .` 에서 첫 점은 나열을, 둘째 점은 `let` 을 닫는다. 원소를 늘어놓는 모양과 섞지 않는다

6d **SIMD 값** — `lit vec n t v₁ … [_] .` 은 `vec n t` 값이다. 원소 규칙은 배열 나열과 같고, 더해 t 는 크기 있는 수(`bool` 이 아니다), n 은 1 부터 16 까지의 2 의 거듭제곱이다(E-LIT-COUNT). 값의 타입은 묶는 자리의 `vec` 타입과 레인 수·원소 종류·폭이 모두 같아야 한다(E-TYPE-LANES · E-TYPE-WIDTH). 레인에 값을 싣는 것으로 끝나므로 (7) 의 자리 규칙과 상관없다.

7 나열이 **어디에 놓이고 얼마나 사는지**는 처리기가 번역 시점에 셋 중 하나로 정한다. 저자는 고르지 않는다. ⓐ `var` 에 묶은 나열과, 실행 중에 계산되는 원소가 든 `let` 나열은 그 op 의 틀 안에 **선언 자리마다 한 칸**을 갖는다. 그 칸은 선언이 실행될 때마다 새로 채워지고(적지 않은 칸은 0), **선언이 든 블록**이 끝날 때까지 산다 — 반복 안의 선언은 바퀴마다 다시 채워지고, 재귀한 부름은 저마다 제 칸을 갖는다. ⓑ 원소가 모두 번역 시점 상수이고 `let` 에 묶였거나 받는 자리 없이 읽기로만 쓰인 나열은 프로그램에 박힌 **읽기 전용** 바이트다. 프로그램 끝까지 살므로 어디에 담아도 된다. 그것을 `mut` 인 이름에 묶거나(`set` 으로 다시 담는 것도 같다) `let` 에 묶은 그 이름을 `mut·owned·mut_ref` 자리에 넘기면 거부된다(`E-TYPE-ARGMUT`). 내용이 같은 두 나열(문자열 리터럴도 같다)이 **같은 자리인지는 자유 선택 (free choice) 이다**(§4.3 · FC-0003) — 처리기가 한 벌로 합칠 수도, 따로 둘 수도 있다. 그러니 두 쪽이 모두 이런 리터럴에서 왔다는 것이 번역 시점에 보이는 `same_slice` 는 거부된다(`E-LIT-IDENTITY` — 리터럴 자신 · 그것에 `let` 으로 묶은 이름 · 그것의 `subslice`). 부름을 건너 보이지 않는 경우는 거부되지 않지만, 그 답에 기대는 프로그램은 적합하지 아니하다. ⓒ 받는 이름 없이 쓰인 나열 가운데 실행 중 원소가 들었거나 `mut·owned·mut_ref` 자리에 곧바로 넘긴 것은 **그 문장이 끝날 때까지** 사는 틀 안 임시다. 받는 op 은 그 칸을 고쳐 써도 된다.

자유 선택 (free choice) — FC-0003 — 내용이 같은 읽기 전용 나열이 같은 자리인가

> 처리기는 내용이 같은 두 읽기 전용 나열(문자열 리터럴도 나열이다)을 **한 벌로 합쳐** 같은 자리에 둘 수도 있고 **따로** 둘 수도 있다. 고를 수 있는 범위는 그 둘뿐이다 — 나열의 내용과 길이, 그리고 프로그램 끝까지 산다는 것은 어느 쪽에서도 같다. 차이가 드러나는 자리는 하나다 — 두 나열이 같은 자리인지를 묻는 `same_slice`. 그래서 두 쪽이 모두 이런 리터럴에서 왔다는 것이 번역 시점에 보이면 그 물음은 거부되고(`E-LIT-IDENTITY`), 보이지 않는 경우에 그 답에 기대는 프로그램은 적합하지 아니하다. 같은 내용인지를 알고 싶으면 내용을 견준다 — 자리를 묻지 않는다.

7a ⓐ·ⓒ 의 칸을 보는 값이 그 수명보다 오래 가는 자리 — `return`, 선언 블록보다 바깥에 선언된 이름(매개변수와 그 칸·원소를 포함한다), 선언 블록보다 바깥에 선언된 actor 에게 보내는 메시지 — 에 담기면 거부된다(`E-LIT-ESCAPE`). 같은 블록이나 더 안쪽에 선언된 actor 에게는 건넬 수 있고, 그 뒤로 그 actor 가 주는 값도 같은 수명을 받는다. ⓒ 임시를 이름에 담는 것도 거부된다 — 나열을 `var` 에 묶어 ⓐ 로 쓴다. 흐름은 부름을 건너 따라간다: 부름의 결과는 그 op 몸에서 결과로 흘러드는 입력의 것만 들고, 스칼라를 돌려주는 부름은 아무것도 들고 나가지 않는다(§8 영역 탈출과 같은 가름이다).

7a-1 **이름 없는 구조체 값**(부름의 결과 · `lit`)은 그 문장 끝에 끝난다. 그 값의 배열 칸을 보는 값을 이름 · `return` · 바깥 자리에 담으면 거부된다(`E-VIEW-TEMP`) — 문장이 끝나면 그 바이트는 다음에 만든 값의 것이다. 그 자리에서 읽기만 하는 것(`len` · `idx` 로 수를 꺼내는 것)은 적합하다. 칸을 오래 보려면 구조체를 먼저 이름에 묶는다(`let b T … .` 뒤 `field b. <칸> .`).

7b `var` 에 묶은 나열은 **쓸 수 있는** 슬라이스다 — 칸은 `set idx buf. i. . v. .` 로 쓴다. 그 이름을 다른 저장소로 다시 묶는 `set buf …` 은 거부된다(`E-ARRAY-SET`). 각 원소는 (6b) 의 폭 규칙을 따르고, 실행 중 원소는 원소 타입과 맞아야 한다(`E-TYPE-SET` · `E-TYPE-WIDTH`). `idx` 와 `subslice` 를 거쳐 닿은 칸도 같은 저장소다 — `set idx subslice buf. a. b. . i. . v. .` 에서 `buf` 가 쓸 수 있는 이름이 아니면 거부된다(`E-TYPE-MUT`). 쓸 수 없는 슬라이스를 `mut slice` 이름에 다시 묶지 못한다 — `let w mut slice u32 g. .` 와 `let w mut slice u32 subslice g. a. b. . .` 는 `g` 가 쓸 수 있는 이름이 아니면 거부된다(`E-TYPE-MUT`) — `mut slice` 인 `var` 에 `set w. g. .` 로 다시 담는 것도 같다. 새 이름은 같은 저장소를 가리키므로, 그것으로 쓰는 것은 `g` 에 쓰는 것이다. 뷰 `view_array <타입> g.` 도 베끼지 않으므로 같다 — 그것을 `mut` 이름에 묶거나, `mut` 입력에 넘기거나, 그것을 거쳐 `set` · `copy` · `store` 하면 `g` 가 쓸 수 있는 이름이어야 한다(`E-TYPE-MUT` · `E-TYPE-ARGMUT`). `copy` 와 `store` 의 받는 쪽이 `subslice` 일 때도 그렇다. `mut` 없는 슬라이스(또는 뷰) 타입의 `var` 는 다른 슬라이스로 다시 묶을 수 있지만 칸을 쓰는 이름이 아니다 — 그 이름으로 칸에 쓰면 `E-TYPE-MUT` 이고, `mut·owned` 입력에 넘기면 `E-TYPE-ARGMUT` 이다(무엇을 담았든 — 읽기 입력도 문자열 리터럴도).

7b-1 **내용 복사** — `copy <받는 쪽> <주는 쪽> .` 는 주는 쪽 슬라이스의 내용을 받는 쪽에 베낀다. 받는 쪽은 쓸 수 있어야 하고(`E-TYPE-MUT`), 두 원소 타입이 같아야 하며(`E-TYPE-SET`), 두 길이가 같아야 한다 — 번

역 시점에 다르다고 알면 거부되고(E-COPY-LEN), 실행 중에 다르면 멈춘다. 같은 줄의 두 조각이 겹쳐도 옳게 베낀다(앞뒤가 겹치는 이동과 같다). 받는 쪽에 쓰는 것이므로 매개변수에 베끼는 fn 은 순수하지 않다(E-EFFECT-PURITY).

7b-2 **맞바꾸기** — swap <슬라이스> <i> <j> . 는 두 칸의 값을 맞바꾼다. 칸에 쓰는 것이므로 그 슬라이스는 쓸 수 있어야 한다 (E-TYPE-MUT — let 나열 · 문자열 · mut 아닌 매개변수 · 번역 때 지은 배열은 쓸 수 없다).

7c 한 op 의 ⓐ·ⓒ 칸을 모두 더한 크기에는 처리기가 밝히는 한도가 있고, 넘으면 거부된다(E-FRAME-SIZE — 조용히 힙으로 옮기지 않는다). 이 처리기의 한도는 운영체제가 있는 기계 65,536 바이트 · 운영체제가 없는 기계 4,096 바이트다. 그런 칸을 가진 op 이 자기 자신에게 (다른 op 을 거쳐서라도) 닿을 수 있으면 거부된다(E-FRAME-RECURSIVE) — 재귀 깊이만큼 틀이 곱해지는데 그 깊이는 번역 시점에 모르므로, 스택이 모자라지 않음을 보일 수 없다. 재귀 안의 나열은 할당기에서 받는다((7d)). 부르는 관계를 따라 가장 깊은 길에서 동시에 살아 있는 ⓐ·ⓒ 칸의 합이 아래 예산을 넘으면 번역이 거부된다(E-STACK-BUDGET) — 진입(아무도 부르지 않는 op)마다 잰다. 태스크로 도는 op(spawn <op> · spawn send 의 처리기)은 그린스레드 예산으로 따로 잰다. 실행 중에는 한 호출 사슬 위에 **동시에 살아 있는** ⓐ·ⓒ 칸과 (7c-1) 의 구조체 칸의 합에도 처리기가 밝히는 예산이 있어, 넘치게 할 op 에는 들어가지 않고 **멈춘다** — 스택이 넘쳐 조용히 망가지지 않는다. 이 처리기의 예산은 운영체제가 있는 기계 4 MiB · 그린스레드 64 KiB · 운영체제가 없는 기계 16 KiB 이고, 모든 뒤끝이 같은 수로 세어 같은 자리에서 멈춘다.

7c-1 **구조체 값의 자리는 그 op 의 틀이다.** 처리기는 op 마다 구조체 값 lit 과 구조체를 돌려받는 부름이 쓸 틀 칸의 수를 번역 때 정한다 — 칸이 모자라는 자리의 값은 §8.1 (3a) 의 풀에 산다. 칸의 몫(칸마다 8 바이트 · 배열 칸은 그 바이트)은 (7c) 의 예산에 든다. 틀이 끝나면 그 값도 끝나고, 돌려주는 값은 부른 쪽의 자리로 **베껴진다**. 틀 칸에는 재귀가 거부되지 않는다 — 깊이만큼 틀이 쌓이고, 실행 중 (7c) 의 예산이 멈춘다. 재귀하지 않는 길에서 구조체 값이 가장 깊은 길을 예산 너머로 밀면 번역이 거부된다 (E-STACK-BUDGET). 참조를 받거나 만드는 op 은 남의 자리에 쓸 수 있으므로 틀 칸을 쓰지 않는다 (그 값은 풀에 산다).

7d **할당기에서 받는 나열** — var <이름> use <할당기> mut slice t lit array n t … else <문장>(lit slice 도 같다)은 그 할당기에 나열의 바이트 수를 청하고(send <할당기> reserve <바이트>), 받으면 그 바이트를 (6) (6c) 대로 채워 이름에 묶고, 못 받으면 else 로 간다(§6.5.1 (5)). 실패를 option 째 들고 가려면 타입을 통째로 적는다 — let <이름> use <할당기> option mut slice t lit … .. else 도 option 도 없거나, lit vec 이면 거부된다 (E-LIT-USING). 바인딩 타입의 원소가 나열의 원소 타입과 다르면 거부된다(E-BIND-ELSE). 바이트는 할당기의 것이라 (7e) 가 아니면 (7a) 의 선언 블록 수명을 받지 않고 할당기의 수명을 따른다: 할당기가 ⓐ 칸을 뒤받침으로 받았다면 (send <할당기> init buf) 그 할당기가 주는 바이트도 그 칸의 수명을 받는다. 구조체 값도 같은 철자로 할당기 바이트에 짓는다: var <이름> use <할당기>. s lit s do … end . else <문장> . 은 size_of s . 바이트를 청해, 받으면 0 으로 채우고 s 의 배치를 얹은 뒤(§8.7.2 의 view 와 같은 표현) 적은 칸을 쓴다. 바인딩 타입은 s 여야 한다(E-BIND-ELSE). **수의 슬라이스 칸**은 할당기 바이트 안에 (주소, 길이) 두 낱말로 놓인다 — C 의 struct { const T *p; size_t n; } 와 같고, 자리는 8 바이트에 맞춘다. 이 배치는 할당기에 짓는 구조체에만 있다: 그런 구조체를 남의 바이트에 view 로 얹거나 size_of 로 크기를 묻는 것은 여전히 거부된다(바이트가 주소가 되면 안 된다). 슬라이스 칸에 준 값이 틀 안 자리를 보면 그 구조체도 그 자리의 수명을 받는다 — keep 으로 남겨도 그 자리보다 오래 나르면 거부된다(E-LIT-ESCAPE). owned · actor · 능력 칸, 원소가 수가 아닌 슬라이스 칸, 배치가 없는 안쪽 구조체를 가진 구조체는 짓지 않는다(E-LIT-UNBUILT) — 그 바이트는 칸과 함께 끝을 맺어야 하기 때문이다.

7e **블록 끝에 돌려준다** — 할당기가 freeing_allocator 를 갖추면(§8.13 (6)), (7d) 로 받은 바이트는 그 이름을 선언한 블록을 나갈 때 할당기에 돌려준다(send <할당기> release <받은 조각>). 블록 끝에 닿을 때 · return · 그 블록 안의 break·continue 모두 그렇다. 한 블록에서 여럿을 받았으면 선언의 거꾸로 돌려주고, 받지 못한 것(else 로 간 것)은 돌려줄 것이 없다. release 의 답은 쓰지 않는다 — 돌려받을지는 할당기의 정책이다. 그래서 이 바이트는 (7a) 의 선언 블록 수명을 받는다: 블록 밖으로 나르면 거부된다

(E-LIT-ESCAPE). `freeing_allocator` 가 아닌 할당기의 바이트는 할당기에 남는다. `panic` 으로 멈출 때는 돌려주지 않는다 — 멈춤은 그 op 을 복구 단위에 넘긴다. 저자는 `drop <이름> .` 으로 더 일찍 돌려줄 수 있다. 그러면 블록 끝에서는 다시 돌려주지 않는다. `drop` 은 그 이름을 선언한 블록에서만 받고(안쪽 블록이면 E-OWN-JOIN), 그 뒤로 이름을 쓰면 거부된다(E-OWN-MOVED).

7f **남긴다** — `var <이름> keep <할당기> … else <문장>` 은 (7e) 를 끈다: 블록을 나가도 돌려주지 않으므로 바이트는 할당기의 수명을 따르고(§6.2.6 (7d)), 블록 밖으로 — `return` 으로도 — 나를 수 있다. 돌려주는 일은 저자가 한다(`send <할당기> release <조각>`). 아니면 할당기가 끝날 때 한꺼번에 돌아간다. `keep <할당기>` 는 `use <할당기>` 의 자리(이름 바로 뒤)에 서고, 나열이나 구조체 리터럴을 받는 바인딩에만 쓴다(E-USING-FORM).

8 `let` 에 묶은 배열 나열은 그 바이트를 **보는 슬라이스**다. 길이는 나열이 정한다.

9 나열의 원소는 바이트 배치가 있는 구조체여도 된다(칸이 모두 크기 있는 수 — §8.7.2 의 `view` 가 설 수 있는 구조체). 원소는 준 값의 사본이고, 나열은 그 바이트 위에 `view_array` 와 같은 보기를 얹은 `slice s` 다. 원소의 칸은 `set field <나열>. <자리> <칸> . v. .` 로 쓴다. 자리 · 수명 · _ · 칸 골라 채우기 · 할당기 절은 수 원소와 같다. 원소 자리의 값이 수이거나 참거짓이면 E-TYPE-FIELD, 다른 구조체이면 E-TYPE-STRUCT 로 거부된다. 이 처리기가 아직 짓지 않은 나열 — 바이트 배치가 없는 구조체 · 칸이 구조체인 구조체 · 줄을 골라 채우기(`do … end`) — 은 무엇이 아직인지 말하며 거부된다(E-LIT-UNBUILT). 구조체 안의 배열 칸은 §6.2.7 (1a) 가 정한다.

9-1 칸이 크기 있는 수 · `bool` 과 **수의 슬라이스**뿐인 구조체도 나열의 원소가 된다. 원소 안에서 슬라이스 칸은 (가리키는 자리, 길이) 두 낱말로 놓인다 — 할당기 바이트에 지은 구조체((7d))와 같은 배치다. 칸은 실행할 때 쓰므로 그런 나열은 ⓐ 의 틀 안 자리에 선다. 같은 원소를 받는 `slice s` 입력과 배열 결과 `output r array n s .` 도 이 배치다. 슬라이스 칸은 남의 바이트를 가리킬 뿐이라, 원소를 베껴도 가리키는 바이트는 베끼지 않는다.

9a **줄의 나열** — 원소가 고정 길이 배열 `(array m t)` 인 나열 `lit slice array m t <줄>… .` 의 타입은 `slice array m t` 이다. 다른 슬라이스 나열처럼((6)) 줄의 수를 적지 않고 _ 가 없다((6a)) — 0 인 줄은 `lit array m t _ .` 로 적는다. 줄 수를 적은 `lit array n array m t <줄>… [_] .` 는 배열 안의 배열의 값이다((1d)). 이미 있는 바이트 위에는 `view_array array m t b.` 로 같은 타입의 줄 보기를 얹는다(§8.7 (7a)). `len` 은 줄 수이고, `idx g. r. .` 는 줄 r 의 바이트를 **보는** `slice t`(길이 m, 복사 없음)다 — 바깥이 `mut` 이면 그 줄의 칸도 쓸 수 있다. 줄 값은 칸마다 베껴지고, 줄 리터럴의 모양이 `array m t` 과 다르면 거부된다(E-LIT-COUNT). 한 줄을 통째로 바꾸는 `set idx g. r. . <줄> .` 은 없다(E-TYPE-SET) — 칸을 쓰거나 `copy` 로 베낀다. 틀 안 나열의 줄(과 구조체 원소)은 그 틀의 바이트를 보므로 블록 밖으로 나르면 E-LIT-ESCAPE 다.

> **예제 (example) — 원소 나열 리터럴**

소스 코드 (source)

```
module ex_list_literal .

fn total input xs slice u32 . output u64 . do
  var s u64 0 .
  for x xs. do
    set s. add s. widen u64 x. . . .
  end .
  return s. .
end .

rem 배열 나열은 제 점으로 닫힌다 — 문장의 끝이면 점이 둘이다.
export fn f output u64 . do
  let t lit array 4 u32 10 20 30 _ . .
  return add total t. . total lit slice u32 1 2 . . . .
end .
```

예제 (example) — var 배열을 반복으로 채운다

소스 코드 (source)

```
module ex_frame_list .

export fn squares input n usize . output u64 . do
  var buf lit array 8 u64 _ . .
  var i usize 0 .
  while lt i. 8 . do
    set idx buf. i. . cast u64 mul i. i. . . .
    set i. add i. 1 . .
  end .
  guard lt n. 8 . else return 0 . .
  return idx buf. n. . .
end .
```

예제 (example) — 칸을 골라 채운다

소스 코드 (source)

```
module ex_cell_fill .

export fn pick input a u64 . input i usize . output u64 . do
  let t lit array 6 u64 do 0 100 . 5 mul a. 2 . . _ 1 . end . .
  guard lt i. 6 . else return 0 . .
  return idx t. i. . .
end .
```

거부되는 예제 (rejected) — 칸 번호를 두 번 적었다

소스 코드 (source)

```
module ex_cell_twice .

export fn f output u64 . do
  let t lit array 4 u8 do 1 1 . 1 2 . _ 0 . end . .
  return len t. . .
end .
```

진단: `E-LIT-INDEX`

거부되는 예제 (rejected) — 블록 안의 var 배열을 돌려준다

소스 코드 (source)

```
module ex_frame_escape .

export fn f output slice u8 . do
  var b lit array 2 u8 1 2 . .
  return b. .
end .
```

진단: `E-LIT-ESCAPE`

거부되는 예제 (rejected) — 원소가 길이보다 적은데 끝에 _ 가 없다

소스 코드 (source)

```
module ex_list_short .

export fn f output u64 . do
  let t lit array 4 u8 1 2 3 . .
  return len t. . .
end .
```

진단: E-LIT-COUNT

> **쉬운 말로**
>
> 포인터만 있는 언어에서는 *"이 포인터가 가리키는 곳에 몇 개가 있는가"*를 사람이 따로 알고 있어야 한다. 그 지식은 소스에 안 적혀 있어서 틀리기 쉽고, 틀리면 남의 메모리를 읽는다. 슬라이스는 그 지식을 **값 안에 넣어** 그 문제를 없앤다.

6.2.6.1 여러 차원의 줄 — `matrix`

1 `matrix k d1 … dk t` 는 타입 `t` 의 값이 `d1 × … × dk` 개 놓인, 차원이 `k` 개인 줄이다. 차원 수와 크기는 타입의 일부이며 번역 시점에 정해진다. 차원 수나 크기가 하나라도 다르면 다른 타입이다 — `matrix 2 3 4 u32` 는 `matrix 2 4 3 u32` 과도, `array 12 u32` 와도 다르다.

1a 차원 수를 먼저, 크기를 차원 수만큼 그다음에, 원소 타입을 맨 뒤에 적는다(§6.2.6 (1a) 와 같은 차례). 차원 수는 2 이상 8 이하이고 크기는 1 이상이며, 모두 꾸밈 없는 십진 리터럴이다(op 의 입력에서는 크기가 번역 시점 값 입력의 이름일 수 있다 — (3a)). 옛 차례(`matrix t k d…` — 원소 타입이 먼저)는 거부된다(E-TYPE-ORDER). 원소 수(크기들의 곱)는 2^32 를 넘지 못한다. 원소 타입에 `matrix` 나 `array` 를 적지 못한다. 어기면 거부된다(E-MATRIX-TYPE).

> **참고 (informative)**
>
> 차원 수를 먼저 적으므로 크기가 어디서 끝나는지 글자만으로 안다(크기는 입력의 이름일 수도 있다). 뒤에 원소가 나열되어도(`lit matrix 2 2 3 u32 1 2 3 4 5 6 .`) 어디까지가 크기이고 어디부터가 원소인지 글자만으로 갈린다.

2 **배치.** 원소는 한 덩어리로 이어 놓으며 줄 우선이다 — 마지막 색인이 가장 빨리 돈다. 크기와 정렬은 `array n t`(`n` 은 크기들의 곱)과 같고, 덧붙는 머리는 없다.

> **참고 (informative)**
>
> 이 배치는 C 의 `t a[d1]…[dk]` 와 같다. `flat m. .` 을 `slice t` 입력을 가진 `extern` op 에 넘기면 C 는 포인터와 원소 수를 받고, 그 포인터를 `t (*)[d2]…[dk]` 로 읽어 같은 칸을 가리킨다.

3 `matrix` 타입은 **이름을 짓는 자리**에 적는다 — `let` · `var` 의 타입, op 의 `input` 과 `output`, 구조체의 칸. 다른 타입의 원소, 타입 인자, 별칭의 자리에는 적지 못한다(E-MATRIX-USE).

3a op 의 `input` 에 한하여, 크기 자리에 그 op 의 번역 시점 값 입력의 이름을 적을 수 있다 — `input comptime r u64 . input comptime c u64 . input m matrix 2 r c u32 .` . 부르는 쪽은 그 입력에 수를 적고(`sum 3 4 a. .`), 인자의 모양은 그 수로 읽은 모양과 같아야 한다(E-MATRIX-SHAPE). `let` · `var` · `output` · 구조체의 칸과 `lit matrix` 의 크기는 수여야 한다(E-MATRIX-TYPE).

> **참고 (informative)**
>
> 크기를 이름으로 받는 입력에는 길이의 진입 계약이 붙지 않는다. 그 자리에 올 수 있는 것은 부르는 쪽에서 모양을 견준 `matrix` 뿐이고, 원소를 읽고 쓰는 것은 (5) 의 검사를 그대로 받는다.

4 **값.** `lit matrix k d1 … dk t <원소>… .` 는 원소를 줄 우선 차례로 나열한다. 원소 수는 크기들의 곱과 같아야 하며, 모자라면 끝에 `_` 를 적어 나머지를 0 으로 채운다. 바인딩에 타입과 리터럴을 함께 적으

면 둘의 모양이 같아야 한다. 어기면 거부된다(E-MATRIX-SHAPE). 구조체 리터럴에서 `matrix` 칸의 값은 그 칸과 같은 모양의 `lit matrix` 다.

5 **읽기와 쓰기.** `idx m. i1 … ik .` 는 색인을 차원마다 하나씩 받는다. 색인 수가 차원 수와 다르면 거부된다 (E-MATRIX-INDEX) — 색인을 덜 준 `idx` 가 줄을 뜻하지는 아니한다(줄은 (7a) 의 `row` 다). 색인은 저마다 **제 차원의 크기**로 검사한다: 어느 하나라도 제 크기 이상이면 실행이 멈춘다. 색인 식은 적힌 차례로 한 번씩 계산한다. `set idx m. i1 … ik . v. .` 는 그 칸에 쓰며, `m` 은 쓸 수 있는 이름이어야 한다(E-TYPE-MUT).

6 `dim m. j .` 는 차원 `j` 의 크기다(0 부터 센다). `j` 는 차원 수보다 작은 꾸밈 없는 십진 리터럴이어야 한다 (E-MATRIX-INDEX). 그 값은 번역 시점의 상수다.

7 `flat m. .` 는 같은 원소를 한 줄로 본 `slice t` 다. 복사하지 아니하며, 길이는 원소 수다.

7a `row m. i1 … i(k-1) .` 는 마지막 차원 하나를 남긴 **줄**이다 — 색인은 마지막을 뺀 차원마다 하나씩이고, 값은 그 줄의 원소를 보는 `slice t` 다. 복사하지 아니하며 길이는 마지막 차원의 크기다. 색인 수가 다르면 거부되고(E-MATRIX-INDEX), 색인은 (5) 와 같이 저마다 제 차원의 크기로 검사한다. `m` 이 쓸 수 있는 이름이면 그 줄을 `mut slice t` 이름에 묶어 쓸 수 있고, 아니면 그렇게 묶지 못한다(E-TYPE-MUT · `let` 으로 묶은 구조체의 칸이면 E-IMMUTABLE). 남는 차원이 둘 이상인 보기(면)는 없다.

7b (5) · (6) · (7) · (7a) 의 `m` 자리에는 `matrix` 로 적힌 이름뿐 아니라 `matrix` 칸을 가리키는 `field x. g .` 도 선다 — `idx field b. g . r. c. ..`

8 `len m. .` 과 `for x m. do … end .` 는 거부된다(E-MATRIX-USE). 차원 하나의 크기는 `dim` 으로, 모든 원소를 도는 것은 `for x flat m. . do … end .` 로 적는다.

9 `matrix` 이름을 통째로 적을 수 있는 자리는 `idx` · `row` · `dim` · `flat` 의 첫 인자, 같은 모양의 입력을 가진 op 의 인자, 같은 모양의 결과를 가진 op 의 `return` 이다. 그 밖의 자리는 거부된다(E-MATRIX-USE). 인자와 입력, 바인딩의 타입과 부른 op 의 결과는 차원 수와 크기가 모두 같아야 한다(E-MATRIX-SHAPE).

9a 결과가 `matrix` 인 op 의 호출은 두 자리에만 놓인다 — 같은 모양의 `matrix` 이름을 짓는 바인딩의 값, 같은 모양의 결과를 가진 op 의 `return` 의 값. 결과가 `matrix` 인 op 의 `return` 은 그 모양의 `matrix` 이름이나 그 모양을 내는 op 의 호출을 준다. `array` 이름에 묶거나 `array` 를 내는 op 에서 돌려주는 것을 포함하여, 어기면 거부된다 (E-MATRIX-SHAPE).

10 이 타입은 **자리**만 정한다. 행렬의 곱 같은 셈은 정하지 아니한다.

> **예제 (example) — 3 줄 4 칸의 표를 채우고 한 줄을 더한다**

소스 코드 (source)

```
module ex_matrix .

fn table output t matrix 2 3 4 u32 .
do
  repeat r usize dim t. 0 . do
    repeat c usize dim t. 1 . do
      set idx t. r. c. . cast u32 add mul r. 10 . c. . . .
    end .
  end .
  return t. .
end .

export fn row_sum input r usize . output u64 .
  requires lt r. 3 . .
do
  let t matrix 2 3 4 u32 table . .
  var s u64 0 .
  repeat c usize dim t. 1 . do
    set s. add s. widen u64 idx t. r. c. . . . .
  end .
```

```
  return s. .
end .
```

예제 (example) — 구조체의 칸인 표, 줄 하나, 어떤 크기든 받는 op

소스 코드 (source)

```
module ex_matrix_more .

struct board do
  id u64 .
  g matrix 2 3 4 u32 .
end .

fn sum input comptime r usize . input comptime c usize . input m matrix 2 r c u32 . output u64 .
do
  var t u64 0 .
  repeat i usize dim m. 0 . do
    for x row m. i. . do set t. add t. widen u64 x. . . . end .
  end .
  return t. .
end .

export fn total output u64 .
do
  var b lit board do id 1 . g lit matrix 2 3 4 u32 1 2 3 4 5 6 7 8 9 10 11 12 . . end . .
  set idx field b. g . 0 0 . 100 .
  let small lit matrix 2 2 2 u32 1 2 3 4 . .
  return add widen u64 idx field b. g . 0 0 . . sum 2 2 small. . . .
end .
```

거부되는 예제 (rejected) — 차원은 둘인데 색인을 하나만 주었다

소스 코드 (source)

```
module ex_matrix_row .

export fn f output u64 . do
  let m lit matrix 2 2 3 u32 1 2 3 4 5 6 . .
  return widen u64 idx m. 1 . . .
end .
```

진단: `E-MATRIX-INDEX`

거부되는 예제 (rejected) — 3 줄 4 칸을 받는 op 에 4 줄 3 칸을 넘긴다

소스 코드 (source)

```
module ex_matrix_shape .

fn first input m matrix 2 3 4 u32 . output u32 . do
  return idx m. 0 0 . .
end .

export fn f output u32 . do
  let m lit matrix 2 4 3 u32 _ . .
```

```
  return first m. . .
end .
```

진단: E-MATRIX-SHAPE

6.2.7 묶음 — struct 와 enum

1 `struct` 는 이름 붙은 칸의 모음이다. 각 칸은 자기 타입을 갖는다.

1a **배열 칸** — 칸의 타입이 `array n t` 이면 그 칸은 원소 `n` 개의 바이트를 **레코드 안에** 가진다(길이가 타입에 있으므로 자리가 정해진다). `t` 는 크기 있는 수나 `bool` 이다 — 구조체나 배열을 원소로 가진 칸은 이 처리기가 아직 짓지 않았다(E-LIT-UNBUILT). 한 구조체의 배열 칸 바이트 합에는 틀과 같은 한도가 있다(E-FRAME-SIZE, §6.2.6 (7c)). `field r. body .` 는 그 바이트를 보는 슬라이스이고, `set idx field r. body . i. . v. .` 가 레코드의 원소를 쓴다. 칸에 주는 나열 리터럴은 칸과 원소 타입·길이가 같아야 한다(E-TYPE-FIELD). 바이트 배치(`view` · `encode` · `size_of` · C 쪽 레이아웃)에서 배열 칸은 C 와 같이 **구조체 바이트 안에 그대로** 놓인다 — 크기는 `n × ¦t¦`, 정렬은 `t` 의 것이다 (`len u8 . body array 4 u8 . tail u16 .` 은 8 바이트 — `len` 은 0 번째 · `body` 는 1 번째 · `tail` 은 6 번째 바이트부터). 뷰의 배열 칸은 그 바이트를 보는 슬라이스이고, 칸에 쓰면 그 자리에 베낀다.

2 `enum` 은 여럿 중 하나다. 각 갈래는 이름을 가지며, 값을 함께 지닐 수 있다.

2a 갈래는 **하나마다 `.` 으로 닫는다.** 개행은 닫개가 아니므로, 점 없이 줄마다 적은 갈래는 한 갈래로 이어지며 거부된다(E-ENUM-DOT). 갈래가 지니는 값은 `<칸 이름> <타입>` 짝으로 적는다 — 짝이 맞지 않으면 거부된다(E-ENUM-FIELD).

2b 열거 값은 수가 아니다 — 갈래의 번호를 수로 쓰려면 `cast` 로 적는다(§6.2.16 (5)).

2c **갈래마다 번호가 있다.** 첫 갈래는 1 이고, 다음 갈래는 앞 갈래의 번호에 1 을 더한 것이다. 갈래 이름 뒤에 정수를 적으면 (`ga 11 .`) 그 갈래의 번호가 되고, 다음 갈래는 거기서 이어 센다. 번호는 선언된 차례대로 **올라야** 한다 — 앞 갈래의 번호보다 크지 않은 번호는 거부된다(E-ENUM-NUMBER). 번호는 값을 지닌 갈래가 없는 열거에만 적는다(E-ENUM-NUMBER). 0 은 첫 갈래의 번호가 아니다 — 번호를 적지 않은 열거에서 0 은 어느 갈래도 아니다.

예제 (example) — 갈래에 번호를 적는다

소스 코드 (source)

```
module ex_enum_numbers .

enum hangul do
  ga 11 .
  na .
  da 101 .
  ra .
end .

fn n input h hangul . output u32 . do
  return cast u32 h. . .
end .

rem 11 + 12 + 101 + 102
fn total output u32 . do
  return add add n ga. . n na. . . add n da. . n ra. . . . .
end .
```

실행 결과 (output)

```
total() = 226
```

거부되는 예제 (rejected) — 번호가 오르지 않는다

소스 코드 (source)

```
module ex_enum_number_falls .

enum level do
  low 5 .
  mid .
  high 6 .
end .
```

진단: `E-ENUM-NUMBER`

2d **열거의 폭.** 갈래에 값을 지니지 않은 열거는 블록 안에 `layout <폭> .` 한 줄을 둘 수 있다. 폭은 `u8` · `u16` · `u32` · `u64` 가운데 하나이고, 그 열거가 바이트 배치(§6.2.13)에 들어갈 때 차지하는 바이트 수다 — 바이트에는 갈래의 **번호**(2c)가 든다. 이 줄은 갈래가 아니다. 폭을 적지 않은 열거에는 바이트 배치가 없다. 다음은 거부된다(`E-ENUM-LAYOUT`): 넷 밖의 폭, 어느 갈래의 번호가 폭에 들어가지 않는 것, 값을 지닌 갈래가 있는 열거, `layout` 줄이 둘인 것. ☞ 폭을 저절로 고르지 않는 까닭: 갈래 하나를 더하는 일이 선이나 파일 너머의 배치를 말없이 바꾸면 안 된다.

예제 (example) — 폭을 적은 열거는 한 바이트에 번호로 놓인다

소스 코드 (source)

```
module ex_enum_layout .

enum kind do
  layout u8 .
  ping .
  data 7 .
  close .
end .

struct hdr do
  layout packed .
  k kind .
  len u16 big .
end .

fn kind_of input b slice u8 . output u8 . do
  let o option view hdr try_view hdr b. . .
  guard is_some o. . else return 0 . .   rem 2 는 어느 갈래의 번호도 아니다
  let h view hdr some_value o. . .
  match field h. k . do
    case ping do return 1 . end .
    case data do return 2 . end .
    case close do return 3 . end .
  end .
end .
```

실행 결과 (output)

```
kind_of([7,0,3]) = 2 · kind_of([2,0,3]) = 0
```

거부되는 예제 (rejected) — 번호가 폭에 들어가지 않는다

소스 코드 (source)

```
module ex_enum_layout_fit .

enum code do
  layout u8 .
  ok .
  far 300 .
end .
```

진단: E-ENUM-LAYOUT

3 struct 는 **자기 자신을 칸으로 가질 수 없다.** 직접이든 다른 타입을 거쳐서든 순환하면 번역이 거부된다. (E-STRUCT-CYCLE). 크기가 정해지지 아니하기 때문이다.

3a 참조를 거쳐 자기를 가리키면 크기는 정해진다. 다만 그 참조가 **누가 관리하는지 알 수 없는 것**이면 거부된다(E-STRUCT-REF-UNMANAGED) — 크기가 있다고 수명이 선 것은 아니다.

3b 칸의 타입이 꼭 `option (owned <그 struct>)` 이면 (3) 의 순환으로 보지 아니한다. 그 칸은 값이 아니라 얼로케이터 바이트에 지은 다른 마디를 가리키는 **주소 한 칸**이고(비어 있으면 none), 그래서 크기가 정해진다. 이런 struct 를 스스로를 품는 마디라 하며, 짓고 돌려주는 규칙은 §8.5 (10)–(13) 이 정한다. `option` 없는 `owned <자기>` 와, 다른 struct 를 거쳐 돌아오는 소유 순환은 여전히 (3) 에 걸린다.

예제 (example) — struct 와 enum

소스 코드 (source)

```
module ex_shape .

struct point do
  x u32 .
  y u32 .
end .

enum color do
  red .
  green .
end .
```

4 struct 값은 `lit` 로 만든다. 만들 때 **모든 칸을 채워야** 한다. 마지막 줄의 `_ <값> .` 은 적지 않은 칸 모두에 그 값을 넣는다(`lit pt do y a. . _ 0 . end .`) — 값은 이름이나 리터럴 하나이고(E-LIT-INDEX), 그 값이 들어갈 수 없는 칸이 남으면 그 칸 이름을 대며 거부되며(E-TYPE-FIELD), 채울 칸이 없으면 거부된다(E-LIT-COUNT).

4a **struct 는 값이다.** 만들거나 베낄 때(`var q point p. .`) 배열 칸의 바이트와, 칸에 든 다른 struct 값까지 함께 베껴진다 — 베낀 쪽을 고쳐도 원본은 바뀌지 아니한다. actor 인스턴스(정체가 있는 것)와 `owned` 칸을 가진 struct 는 베끼지 않고 같은 것을 가리킨다.

4b **칸에 쓰는 것은 그 레코드에 쓰는 것이다.** `set field r. x . v. .` 와 배열 칸의 원소 쓰기는 `r` 이 쓸 수 있는 자리일 때만 받는다: `var` 이거나 `mut` 매개변수다. `let` 으로 묶은 레코드에 쓰면 거부된다(E-IMMUTABLE) — 배열 칸을 쓸 수 있는 슬라이스(`mut slice`)로 꺼내는 것도 같다. 값으로 받은 매개변수의 칸에 쓰면 그 op 의 **지역 복사**에 쓴다 — 부른 쪽의 레코드는 바뀌지 아니한다(§6.5.1 (2a)).

5 칸을 읽을 때는 `field <값> <칸 이름>` 을 쓴다. 값 뒤에 점과 칸 이름을 붙이는 모양은 없다 (E-FIELD-GLUED) — 점은 모듈 한정·갈래 이름에 이미 쓰인다.

예제 (example) — struct 를 만들고 읽는다

소스 코드 (source)

```
module ex_make .

struct point do
  x u32 .
  y u32 .
end .

export fn origin output point .
do
  return lit point do x 0 . y 0 . end . .
end .

export fn get_x input p point . output u32 .
do
  return field p. x . .
end .
```

예제 (example) — 배열 칸을 가진 struct 를 베낀다

소스 코드 (source)

```
module ex_struct_array .

struct pkt do
  len u8 .
  body array 4 u8 .
end .

export fn copy_keeps input a u8 . output u64 . do
  var p lit pkt do len 2 . body lit array 4 u8 1 a. _ . . end . .
  var q pkt p. .
  set idx field q. body . 0 . 100 .
  return add widen u64 idx field p. body . 0 . . widen u64 idx field q. body . 0 . . . .
end .
```

거부되는 예제 (rejected) — let 으로 묶은 레코드의 배열 칸에 쓴다

소스 코드 (source)

```
module ex_struct_array_let .

struct pkt do
  body array 4 u8 .
end .

export fn f output u64 . do
  let p lit pkt do body lit array 4 u8 _ . . end . .
  set idx field p. body . 0 . 1 .
  return 0 .
end .
```

진단: `E-IMMUTABLE`

거부되는 예제 (rejected) — 갈래를 점으로 닫지 않았다

소스 코드 (source)

```
module ex_enum_dot .

enum color do
  red
  green
end .
```

진단: E-ENUM-DOT

참고 (informative)

순환을 금지하는 이유는 크기 때문이다. 자기를 품는 struct 는 크기가 무한해진다. 나무 같은 자료 구조가 필요하면 **번호(색인)** 로 잇는다 — 그러면 크기가 정해지고, 경계 검사가 그대로 성립한다.

6.2.8 답을 담는 타입 — result 와 option

1 `result t e` 는 성공한 값(t) 또는 오류(e) 중 하나를 담는다.

2 `option t` 는 값이 있거나 없음을 담는다.

3 이 두 타입의 값은 **꺼내기 전에 어느 쪽인지 확인해야 한다.**

3a 꺼내는 연산은 **부분 연산**이다 — 확인하지 않고 꺼내는 것 자체는 번역이 거부하지 아니하며, 실행 중에 없는 쪽을 꺼내면 **트랩한다**. 곧 확인은 번역이 대신해 주는 것이 아니라 저자가 하는 것이다.

3b 담긴 값의 타입은 그 타입의 일부다. `option usize` 와 `option u64`, `result u32 e` 와 `result u64 e` 는 서로 다른 타입이며, 한쪽을 다른 쪽의 자리에 대면 번역이 거부된다(E-TYPE-NOMINAL · E-TYPE-WIDTH · E-TYPE-SIGN). 꺼낸 값 (some_value · ok_value)의 타입은 담긴 값의 타입이다. some · ok 가 리터럴을 감쌀 때는 받는 자리의 타입을 따른다.

3c result · option 의 값은 셈이나 견줌(eq · lt …)의 피연산자가 아니다(E-TYPE-KIND). 무엇을 묻는지 적는다 — is_some · is_ok · value_or.

예제 (example) — result 로 실패를 돌려준다

소스 코드 (source)

```
module ex_result .

enum err do
  too_small .
end .

rem 2 보다 작으면 반으로 나눌 수 없다고 알린다.
export fn half input n u32 . output result u32 err .
  errors too_small lt n. 2 . .
do
  guard ge n. 2 . else return error too_small . . .
  return ok div n. 2 . . .
end .
```

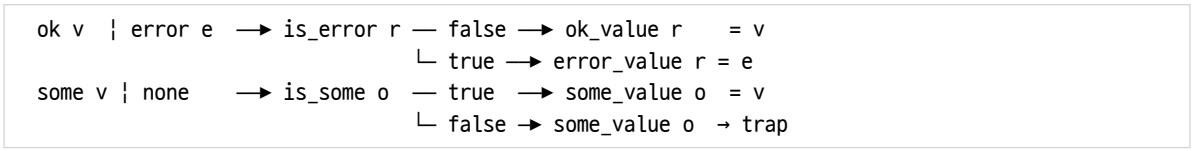
쉬운 말로

실패를 나타내려고 −1 이나 널 포인터 같은 **특별한 값**을 쓰는 관습이 있다. 그러면 "이 *−1* 은 오류인가 그냥 *−1* 인가"를 소스만 보고는 알 수 없다. result 와 option 은 그 물음을 타입으로 옮겨서, 처리기가 대신 물어보게 만든다.

4 만드는 쪽은 ok(성공) error(오류) some(값이 있음) none(값이 없음) 으로 값을 싼다.

5 받는 쪽은 먼저 어느 쪽인지 묻는다 — is_error(오류인가) is_some(값이 있는가). 그 다음 꺼낸다 — ok_value(성공 값) error_value(오류) some_value(있는 값).

도해 (diagram) — 싸고 · 묻고 · 꺼낸다



6 `value_or` 는 값이 있으면 그 값을, 없으면 대신 줄 값을 낸다. 묻고 꺼내는 두 걸음을 한 걸음으로 줄인다.

7 `try` 는 `result` 를 받아 성공하면 값을 꺼내고 실패하면 그 오류를 **그대로 위로 넘긴다**(§6.5.7).

8 `result` 를 받고 **아무도 그것을 보지 아니하면** 처리기는 알린다 (`W-RESULT-DISCARD`). 실패를 값으로 돌려준다는 설계는 받는 쪽이 그 값을 **볼 때만** 지켜진다. 보지 않는 모양은 둘이다 — op 을 **문장으로** 불러 값을 이름조차 없이 버리는 것, 그리고 이름에 **담아 두고 한 번도 읽지 아니하는** 것.

8a 실패를 **일부러 넘기는** 자리는 `drop <이름> .` 으로 적는다. 그러면 알리지 아니한다. 실패로 할 일이 정말 없는 자리가 있다 — 오류 경로에서 자원을 닫고 나가는 자리가 그렇다. `drop` 은 이미 «나는 이것을 여기서 끝낸다» 를 뜻하므로, 그 자리에 새 낱말을 두지 아니한다.

쉬운 말로

왜 거절이 아니라 알림인가. 실패를 무시하는 것이 **언제나** 잘못은 아니다 — 잘못은 **말없이** 무시하는 것이다. 그래서 이 조항이 요구하는 것은 «무시하지 말라» 가 아니라 «무시한다고 적어라» 이고, 적어 두면 읽는 사람이 그것을 **저자의 판단**으로 읽는다. 적히지 않은 것은 판단인지 실수인지 아무도 모른다.

예제 (example) — 확인하지 않고 꺼내면 실행 중에 멈춘다

소스 코드 (source)

```
module ex_partial .

fn mk input k u8 . output option u8 .
do
  guard lt k. 3 . else return none . .
  return some mul k. 10 . . .
end .

rem 확인 없이 바로 꺼낸다 — 번역은 통과한다.
fn raw input k u8 . output u8 .
do
  return some_value mk k. . . .
end .
```

실행 결과 (output)

```
raw(2) = 20 · raw(7) → E-VM-NONE (트랩)
```

쉬운 말로

`raw(2)` 는 20 을 낸다. `raw(7)` 은 값이 없는데 꺼내므로 **멈춘다** — 조용히 0 을 내지 아니한다. 번역이 이것을 미리 막지 않는 까닭은, 어느 길에서 값이 있는지는 **저자가 아는 것**이고 처리기가 언제나 알 수는 없기 때문이다. 대신 틀렸을 때 **조용하지 않다.**

예제 (example) — option — 만드는 쪽과 받는 쪽

소스 코드 (source)

```
module ex_option .

rem 만드는 쪽 — 값이 있으면 some, 없으면 none.
fn lookup input k u8 . output option u8 .
do
  guard lt k. 3 . else return none . .
  return some mul k. 10 . . .
end .

rem 받는 쪽 ① — 묻고 꺼낸다.
fn use_ask input k u8 . output u8 .
do
  let r option u8 lookup k. . .
  guard is_some r. . else return 255 . .
  return some_value r. . .
end .

rem 받는 쪽 ② — 없으면 대신 쓸 값을 준다.
fn use_or input k u8 . output u8 .
do
  return value_or lookup k. . 99 . .
end .
```

실행 결과 (output)

```
use_ask(2) = 20 · use_ask(7) = 255 · use_or(2) = 20 · use_or(7) = 99
```

쉬운 말로

`output option u8 .` 의 점은 하나다 — `option` 은 타입을 **하나** 받으므로 그 인자 수에서 끝나고, 점은 `output` 절을 닫는다. 타입마다 점을 겹쳐 적던 옛 모양(`option u8 . .`)은 거부된다(§6.1.6 (2e)).

예제 (example) — result — 묻고 꺼내기, 그리고 `try` 로 넘기기

소스 코드 (source)

```
module ex_result_use .

enum io_error do
  too_big .
end .

rem 만드는 쪽 — 언제 실패하는지 계약으로 적는다.
fn halve input a u8 . output result u8 io_error .
  errors too_big gt a. 200 . .
do
  guard le a. 200 . else return error too_big . . .
  return ok div a. 2 . . .
end .

rem 받는 쪽 ① — 묻고 꺼낸다.
fn use_ask input a u8 . output u8 .
do
  let r result u8 io_error halve a. . .
  guard not is_error r. . . else return 0 . .
  return ok_value r. . .
end .
```

```
rem 받는 쪽 ② — try 는 실패를 그대로 위로 넘긴다.
fn use_try input a u8 . output result u8 io_error .
  errors too_big gt a. 200 . .
do
  let v u8 try halve a. . . .
  return ok add v. 1 . . .
end .
```

실행 결과 (output)

```
use_ask(100) = 50 · use_ask(250) = 0 · use_try(100) = ok 51
```

쉬운 말로

`use_try` 가 `errors` 를 자기도 적은 것에 주의한다. `try` 로 실패를 넘기려면 **자기도 그 오류를 돌려줄 수 있어야** 하고, 그것을 계약에 적어야 한다. 실패가 조용히 사라지는 길이 없다.

6.2.9 이름 붙인 타입

1 `type` 은 기존 타입에 다른 이름을 준다. 두 이름은 **같은 타입**이며 서로 바꿔 쓸 수 있다.

2 `newtype` 은 기존 타입과 같은 표현을 갖되 **다른 타입**을 만든다. 서로 바꿔 쓸 수 없다.

2a 모양은 `type <이름> <타입> .` 과 `newtype <이름> <타입> .` 이다. 이름과 타입 사이에 낱말을 끼우지 아니한다.

거부되는 예제 (rejected) — `newtype` 은 바탕 타입과 섞이지 아니한다 — (2) 를 시험한다

소스 코드 (source)

```
module ex_newtype_mix .

newtype node_id u64 .

fn f input a node_id . input b u64 . output u64 . do
  return add a. b. . .
end .
```

진단: `E-TYPE-NOMINAL`

2b `type` 과 `newtype` 의 타입은 **줄 타입** — `slice` · `array` · `matrix` — 그 자체일 수 없다. `mut` 를 앞에 붙인 것도 같다. 줄 타입은 쓰는 자리에 그대로 적는다. 다른 타입 **안에** 든 줄 타입은 이름을 붙일 수 있다 (`type found option slice u8 .`). 제 이름을 가진 타입이 필요하면 구조체의 칸으로 둔다(§6.2.7).

표 21 — 줄 타입에 이름을 붙였을 때의 진단

어긴 것	진단
(2b) — 타입 전체가 `slice`	`E-TYPE-ROWNAME`
(2b) — 타입 전체가 `array`	`E-TYPE-ARRAY`
(2b) — 타입 전체가 `matrix`	`E-MATRIX-USE`

거부되는 예제 (rejected) — 슬라이스에 다른 이름을 붙였다 — (2b) 를 시험한다

소스 코드 (source)

```
module ex_rowname .

type bytes slice u8 .

fn f input b bytes . output u64 . do
  return len b. . .
end .
```

진단: `E-TYPE-ROWNAME`

쉬운 말로

줄 타입은 원소의 타입과 길이 · 차원이 낱말에 그대로 드러난다. 다른 이름 뒤에 숨기면 읽는 사람이 선언을 찾아가야 그것을 안다 — 같은 타입을 적는 길도 둘이 된다. 구조체의 칸은 다르다: 이름이 다르면 타입이 다르므로 그 이름은 숨기는 것이 아니라 **가르는** 것이다.

참고 (informative)

둘을 가르는 이유는 실수를 막기 위해서다. 사용자 번호와 주문 번호가 둘 다 `u64` 라면 바꿔 넣어도 처리기가 모른다. `newtype` 으로 갈라 두면 처리기가 그 실수를 잡는다.

3 `newtype` 이 **저장소를 가르는 이름**(상표 (brand))으로 쓰일 때, 한 상표는 저장소 **하나**를 가리킨다. 같은 상표로 저장소를 두 번 여는 것은 거부된다 (`E-BRAND-REUSED`) — 그러면 한 이름이 둘을 가리키게 되고, 서로 다른 저장소에서 온 손잡이가 **같은 타입으로 보인다.** 그것은 상표가 막으려던 바로 그 혼동이다. 갈라야 한다면 `newtype` 을 하나 더 선언한다.

6.2.10 값의 안을 읽는다 — field

1 `field` 는 값의 안에 있는 것을 읽는 폼이다. 모양은 `field <값> <마디> …` 이며, 마디는 **하나 이상**이다. 마디가 여럿이면 왼쪽에서 오른쪽으로 한 칸씩 내려간다 — `field o. inner deep .` 은 `o` 의 `inner`, 그 안의 `deep` 을 뜻한다.

2 마디는 두 가지 중 하나다. **이름**이면 struct 의 필드이고, **정수**이면 줄(배열·슬라이스)의 자리다. 둘은 섞일 수 있다.

3 `field` 는 자리 (place) 이기도 하다 — `set` 의 왼쪽에 올 수 있다. 읽는 자리와 쓰는 자리가 **같은 철자**를 갖는다.

4 선언에 없는 필드를 적으면 처리기는 프로그램을 거절한다. 줄의 자리는 §6.2.6 의 경계 규칙을 따른다.

예제 (example) — 다단 필드 읽기와 쓰기

소스 코드 (source)

```
module ex_field .

struct inner do
  a u64 .
end .

struct outer do
  i inner .
end .

export fn read_deep input o outer . output u64 .
do
  return field o. i a . .
```

```
end .

export proc bump_deep input o mut outer .
  effects state .
do
  set field o. i a . 1 .
end .
```

참고 (informative)

점을 붙여 적는 표기(`o.i.a`) 는 **없다**. 값에 대한 연산을 이름 경로처럼 보이게 하면, 읽는 사람이 `o` 가 값인지 모듈인지 알아야만 뜻이 정해진다 — 그리고 그것은 문법이 아니라 이름 해소의 일이다. 폼으로 적으면 **읽는 순간에** 정해진다.

6.2.11 레인을 가진 값 — `vec`

1 `vec <레인 수> <원소 타입>` 은 같은 타입의 값 여럿을 **한 값으로** 든다(§6.2.6 (1a) 와 같은 차례 — 레인 수가 먼저다. 옛 차례 `vec <원소 타입> <레인 수>` 는 `E-TYPE-ORDER`). 레인 수 자리에는 `scalable` 도 선다(`vec scalable u8`). 레인 수는 번역 시점에 정해진 값이어야 한다(§6.8).

2 레인 수가 다른 두 `vec` 은 서로 다른 타입이며 섞이지 아니한다.

3 `vec` 에 대한 산술은 **레인마다** 따로 일어난다. 레인 사이에 값이 넘나들지 아니한다.

3a 레인마다의 산술은 `add` · `sub` · `mul` · `div` · `mod` · `neg` · `abs` · `min` · `max` 이다. 정수 레인의 넘침은 그 레인의 폭으로 **감긴다**(스칼라와 다르다 — 부호 있는 레인의 가장 작은 값을 -1 로 나눈 몫과 그 `neg` · `abs` 는 그 값 자신이다). 정수 레인의 `div` · `mod` 에서 나누는 수가 0 인 레인이 있으면 스칼라와 같이 멈춘다. 몫은 0 쪽으로 자르고, `mod` 는 나누는 수의 부호를 따르며(§6.3), 부호 있는 레인의 `min` · `max` 는 부호로 견준다(2026-11-02). 부동소수점 레인에는 `mod` 가 없다 — 스칼라와 같이 번역이 거부된다(§6.3.3 (2a) · `E-TYPE-KIND`).

참고 (informative)

기계가 여러 레인을 한 번에 셈할 수 있으면 그렇게 하고, 못 하면 하나씩 셈한다. **어느 쪽이든 답은 같다** — 레인 수는 성능의 힌트가 아니라 타입의 일부다.

거부되는 예제 (rejected) — 레인 수를 원소 타입 뒤에 적었다

소스 코드 (source)

```
module ex_vec_order .

export fn sum4 input xs slice u32 . output u32 .
  requires ge len xs. . 4 . .
do
  let v vec u32 4 load xs. 0 . .
  return reduce_add v. . .
end .
```

진단: `E-TYPE-ORDER`

6.2.12 비트로 이루어진 집합 — `bitset`

1 `bitset <비트 수>` 는 0 부터 그 수 미만까지의 값이 들어 있는지 없는지를 담는다. 비트 수는 번역 시점에 정해진 값이어야 한다.

2 범위 밖의 값을 넣거나 묻는 것은 적합하지 아니하다.

3 `bitset` 은 **집합**이지 워드의 비트가 아니다. 워드의 비트를 다루는 것은 비트 연산이며 (§6.3), 둘은 서로 다른 것이다.

4 원소를 넣는 것은 `bitset_insert <집합> <값> .`, 지우는 것은 `bitset_remove <집합> <값> .` 이다 — 둘 다 그 자리에서 집합을 바꾼다. `add` 는 수의 덧셈만 뜻하므로 집합에 쓰는 것은 적합하지 아니하다(E-BITSET-ADD).

4a 집합에 든 원소의 수는 `bitset_count <집합>` 이고, 그 타입은 `usize` 다. 스택의 원소 수는 `stack_count <스택>` 이다. 둘을 서로 바꿔 쓰거나 조각의 길이인 `len` 을 집합 · 스택에 쓰면 번역이 거부된다(E-TYPE-ARG).

> **예제 (example) — 집합에 넣고 지운다**
>
> **소스 코드 (source)**
>
> ```
> module ex_bitset .
>
> fn members output usize . do
>   var s bitset 8 bitset_new 8 . .
>   bitset_insert s. 3 .
>   bitset_insert s. 5 .
>   bitset_remove s. 3 .
>   return bitset_count s. . .
> end .
> ```
>
> **실행 결과 (output)**
>
> ```
> members() = 1
> ```

6.2.13 바이트 위에 얹는 눈 — 뷰

1 바이트 슬라이스를 **복사 없이** 다른 타입의 배열처럼 읽을 수 있다. 이것을 뷰(view)라 한다.

2 뷰를 얻을 때 정렬과 길이의 계약이 확인된다. 계약이 깨지면 프로그램이 멈춘다.

3 뷰는 바이트를 **복사하지 아니한다.** 뷰를 통해 쓰면 원래 바이트가 바뀐다.

> **주의 — 뷰는 길이를 원소 수로 센다**
>
> 바이트 슬라이스의 길이는 바이트 수이지만, 그것을 네 바이트짜리 원소의 배열로 보면 길이는 **원소 수**다. 같은 저장소를 두 눈으로 보면 `len` 이 다르게 답한다 — 다른 것을 세고 있기 때문이다.

4 뷰의 반대는 `encode` 다. `encode <짜임> <값>` 은 그 값을 짜임이 정한 배치대로 **바이트에 적는다.** 얹어 읽는 것이 `view` 라면, 적어 내리는 것이 `encode` 다.

5 **구조체 위에 얹는 뷰는 창이다.** `view <구조체> <바이트>` 는 바이트 슬라이스 위에 그 구조체의 배치를 얹은 **창**을 준다. 창의 칸을 읽으면 그 자리의 바이트를 읽고, 칸에 쓰면 그 자리의 바이트에 쓴다 — 베낀 사본이 아니다. 창의 타입은 `view T`(읽기 창) 또는 `mut view T`(쓰기 창)이고, 구조체 값의 타입 `T` 와 다르다. `try_view` 는 같은 일을 하되 길이 · 정렬이 맞지 않거나 칸이 맞지 않으면(9a) `none` 을 준다 — 그 타입은 `option view T` · `option mut view T` 이고 `some_value` 는 창이다. 길이와 정렬의 조건은 §8.7.2 가 정한다.

6 **창과 값은 섞이지 않는다.** 창을 구조체 이름 `T` · `option T` 의 자리에 두면 — 묶기 · 돌려주기 · 창 이름에 구조체 값을 통째로 넣기 — 번역이 거부된다(E-VIEW-COPY). 바이트와 따로 사는 **사본**은 `cast T <창> .` 으로 짓는다(§6.2.16 — 명시적 변환은 `cast` 다). 창을 다른 창 이름에 묶으면 같은 바이트를 본다. 값을 받는 입력(`input p T .`)에는 창도 넘길 수 있다 — 받는 쪽은 읽기만 한다.

표 22 — 창과 값

적은 것	무엇인가
`let h view pt view pt b. .`	`b` 위의 읽기 창. 칸을 읽으면 `b` 를 읽는다

`let h mut view pt view pt b. .`	`b` 위의 쓰기 창. `set field h. y . 9 .` 는 `b` 를 바꾼다 — `b` 가 쓸 수 있어야 한다
`let k mut view pt h. .`	같은 바이트를 보는 또 하나의 쓰기 창
`var k pt cast pt h. . .`	`h` 의 칸을 베낀 구조체 값. `k` 에 써도 `b` 는 그대로다
`put h. .` (`put` 의 입력이 `mut pt`)	`b` 의 바이트. 받는 쪽이 칸에 쓰면 `b` 가 바뀐다 — 쓰기 창이어야 한다
`output view pt` · `output mut view pt`	입력의 바이트를 보는 창을 돌려준다

7 **고칠 수 있음은 타입에 적힌다**(§8.8 (2)). 쓰기 창은 쓸 수 있는 바이트에서만 나온다 — 공유 입력, `mut` 없이 묶은 슬라이스, 문자열 리터럴, 읽기 창, 읽기 창을 내는 op 의 결과에서 쓰기 창을 묶으면 거부된다(E-TYPE-MUT). 읽기 창으로 칸에 쓰거나 `write_volatile` 하면 E-TYPE-MUT, 읽기 창을 `mut` 입력에 넘기면 E-TYPE-ARGMUT, 출력이 `mut view T` 인데 읽기 전용 바이트의 창을 돌려주면 E-TYPE-RETMUT 이다. `fn` 이 `mut` 으로 받은 바이트의 창에 쓰면 — 직접 얻은 것이든 다른 op 이 돌려준 것이든 — 부른 쪽에 보이므로 순수하지 않다(E-EFFECT-PURITY, §7.3 (6)).

8 뷰를 얻을 수 있는 구조체는 **바이트 배치가 정해진** 구조체다. 칸마다 폭이 정해진 수(`u8`…`u64` · `i8`…`i64` · `f32` · `f64`), `bool`, 폭을 적은 열거(§6.2.7 (2d)), 그런 것들의 고정 배열, 또는 그런 칸으로 된 구조체여야 한다. 폭을 적지 않은 열거 · 슬라이스 · `option` · 소유 핸들 칸은 정해진 바이트가 없으므로 그 구조체에는 뷰도 `encode` 도 쓸 수 없다(E-IR-UNDEF).

9 `view` 와 `view_array` 로 읽는 구조체의 칸은 모두 `plain` 이어야 한다(§8.7.1 (2)). `bool` 칸은 $0 \cdot 1$ 아닌 바이트를 받으면 참도 거짓도 아닌 값이 되고, 열거 칸은 어느 갈래의 것도 아닌 번호를 받으면 어느 갈래도 아닌 값이 되므로, 그런 칸이 (안에 든 구조체까지) 하나라도 있으면 거부된다(E-TYPE-BITCAST) — `bit_cast bool` 을 거부하는 것과 같은 까닭이다. `view_array bool` 도 같다. 값을 바이트로 적어 내리는 `encode` 에는 이 제한이 없다.

9a `try_view` 는 그런 칸을 **잰다** — `bool` 칸의 바이트가 $0 \cdot 1$ 이 아니거나 열거 칸의 번호가 그 열거의 어느 갈래의 번호도 아니면 `none` 을 준다. 안에 든 구조체의 칸도 잰다. 그렇게 얻은 값의 열거 칸을 읽으면 그 번호의 갈래가, 열거 칸에 쓰면 그 갈래의 번호가 바이트에 든다. 잰 뒤에 아래 바이트가 다른 길로 바뀌어 어느 갈래의 번호도 아니게 되면 그 칸을 읽는 순간 실행이 멈춘다.

> **예제 (example) — 뷰의 칸에 쓰면 바이트가 바뀌고, 이름에서 이름으로 묶으면 베낀다**
>
> **소스 코드 (source)**

```
module ex_struct_view .

struct pt do
  x u8 .
  y u8 .
end .

proc write_through input b mut slice u8 . output u8 . effects none . do
  let h mut view pt view pt b. . .
  set field h. y . 9 .     rem b 가 [1,9] 가 된다
  return idx b. 1 . .
end .

proc copy_by_name input b mut slice u8 . output u8 . effects none . do
  let h view pt view pt b. . .
  var k pt cast pt h. . .  rem 구조체 값의 사본
  set field k. y . 7 .
```

```
  return idx b. 1 . .      rem b 는 그대로다
end .
```

실행 결과 (output)

```
write_through([1,2]) = 9 · copy_by_name([1,2]) = 2
```

거부되는 예제 (rejected) — 읽기 전용으로 받은 바이트에서 쓰기 창을 묶는다

소스 코드 (source)

```
module ex_struct_view_ro .

struct pt do
  x u8 .
  y u8 .
end .

proc f input b slice u8 . output u8 . effects none . do
  let h mut view pt view pt b. . .  rem 칸에 쓰면 부른 쪽의 b 가 바뀐다
  set field h. y . 9 .
  return 0 .
end .
```

진단: `E-TYPE-MUT`

거부되는 예제 (rejected) — 창을 구조체 값의 자리에 묶는다

소스 코드 (source)

```
module ex_struct_view_copy .

struct pt do
  x u8 .
  y u8 .
end .

fn f input b slice u8 . output u8 . do
  let h pt view pt b. . .  rem 창인가 사본인가 — `view pt` 또는 `cast pt`
  return field h. y . .
end .
```

진단: `E-VIEW-COPY`

거부되는 예제 (rejected) — `bool` 칸은 0 · 1 아닌 바이트를 받을 수 있다

소스 코드 (source)

```
module ex_struct_view_bool .

struct flag do
  active bool .
  n u8 .
end .

fn f input b slice u8 . output u8 . do
  let h view flag view flag b. . .
  return field h. n . .
end .
```

> 진단: `E-TYPE-BITCAST`

6.2.14 값의 범위를 좁힌 매개변수 — `range`

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

2 부르는 쪽이 그 범위 안임을 증명하지 못하면 번역이 거부된다. 프로그램 **바깥**에서 들어온 값(진입점의 인자 따위)은 증명할 부르는 쪽이 없으므로, **경계에서** 확인되며 범위를 벗어나면 프로그램이 멈춘다.

2a 부르는 쪽의 타입이 그 범위보다 **넓으면** 그것만으로 거부된다(`E-TYPE-WIDTH`). 폭이 같아 그 검사에 걸리지 않는 값은 **op 의 진입에서** 범위를 확인하며, 벗어나면 멈춘다 (`E-VM-CONTRACT`) — `requires ge <이름> <아래> .` 와 `requires le <이름> <위> .` 를 적은 것과 같은 검사다.

3 선언한 범위는 §6.4 의 계약과 같은 자격으로 쓰인다 — 검사기가 그것을 사실로 삼아 뒤따르는 검사를 지울 수 있다.

> **쉬운 말로**
>
> (3) 이 (2a) 를 **필요하게 만든다**. 검사기가 범위를 사실로 삼아 경계 검사를 지우는데 그 사실을 아무도 지키지 않으면, 지워진 검사 자리로 범위 밖의 값이 지나간다. 2026-09-16 까지 폭이 같은 `range` 매개변수는 어느 층에서도 확인되지 않았다(결함 노트 10) — **믿을 것은 검사되는 것뿐이다.**

> **거부되는 예제 (rejected) — 부르는 쪽의 타입이 범위보다 넓다**
>
> **소스 코드 (source)**
>
> ```
> module ex_range_wide .
>
> fn scale input a range 0 100 . output u8 .
> do
>   return narrow u8 a. . .
> end .
>
> fn call_scale input x u64 . output u8 .
> do
>   return scale x. . .    rem `u64` 는 0 … 100 보다 넓다 — 범위 안임을 보이지 않았다
> end .
> ```
>
> 진단: `E-TYPE-WIDTH`

> **참고 (informative)**
>
> 같은 일을 `requires` 로도 적을 수 있다. 다른 점은 **어디에 적히는가**다. 범위를 타입 자리에 적으면 그 제약이 매개변수의 **생김새**가 되어, 부르는 쪽이 서명만 보고도 안다. 계약절로 적으면 계약을 읽어야 안다.

6.2.15 바이트 배치를 못 박는다 — `layout` 과 `align`

1 구조체의 선언 안에 배치를 적을 수 있다. 적지 아니하면 필드는 **선언 차례대로** 놓이고, 각 필드는 제 타입의 정렬에 맞춰 놓인다(그 사이의 채움은 정렬이 요구하는 만큼). 처리기는 차례를 바꾸지 아니한다.

2 `layout packed .` 은 필드 사이에 **채움(padding)을 두지 아니한다** — 필드가 선언 차례로 맞붙는다.

2a 배치의 낱말은 닫힌 둘이다 — `packed` 와 `native`. 그 밖은 거부된다 (`E-TYPE-LAYOUT`).

3 `align <n> .` 은 그 구조체가 시작하는 자리의 정렬을 못 박는다. n 은 2 의 거듭제곱이어야 한다. 정렬은 **2 의 거듭제곱**이어야 한다(`E-TYPE-ALIGN`) — 그 밖의 수는 어떤 기계도 지킬 수 없다.

> **표지** (marker)
>
> 필드 뒤에 붙여 그 필드가 **어떻게 놓이고 어떻게 닿을 수 있는지**를 정하는 낱말. 타입이 아니므로 값의 크기를 바꾸지 아니하며, 바이트 순서(byte order)와 닿는 법만 정한다.

4 필드 뒤에는 표지 (marker) 를 붙일 수 있다. 표지는 다음 다섯이 전부이며, 그 밖의 낱말을 적으면 번역이 거부된다(`E-FIELD-MARK`).

표 23 — 필드 표지 — 닫힌 집합 다섯

표지	무엇을 정하나	뜻
`big`	바이트 순서	**빅 엔디언(big-endian)**으로 놓는다
`little`	바이트 순서	**리틀 엔디언(little-endian)**으로 놓는다
`rw`	닿는 법	읽고 쓸 수 있다
`ro`	닿는 법	**읽기만** 한다 — 쓰면 번역이 거부된다
`wo`	닿는 법	**쓰기만** 한다 — 읽으면 번역이 거부된다

4a 바이트 순서를 적지 아니하면 그 기계의 순서를 따른다(§5.6).

4b 닿는 법은 **장치 레지스터를 비추는 구조체**(§7.6)에서만 강제된다. 어긴 것은 실행할 때가 아니라 **번역할 때** 거부된다(`E-MMIO-PERM`).

> **쉬운 말로**
>
> `wo` 를 읽는 것이 왜 오류인가. 쓰기 전용 레지스터는 읽으면 쓰레기가 나오거나, 읽는 행위 자체가 장치를 움직인다. 곧 그 읽기는 **쓸모없는 것이 아니라 틀린 것**이다. 장치가 무엇을 받아들이는지 말했으면, 타입이 그것을 되말한다.

> **예제 (example) — 선으로 나가는 머리 — 채움 없이, 빅 엔디언으로**
>
> **소스 코드 (source)**
>
> ```
> module ex_layout .
>
> struct wire_header do
>   layout packed .
>   magic u32 big .
>   length u16 big .
>   kind u8 .
> end .
>
> fn hdr_kind input b slice u8 . output u64 . do
>   let v view wire_header view wire_header b. . .
>   return widen u64 field v. kind . . .
> end .
> ```

> **참고 (informative)**
>
> **배치를 못 박는 것과 안 박는 것은 다른 값을 산다.** 안 박으면 처리기가 기계에 맞게 고를 수 있어 빠르다. 못 박으면 그 바이트가 **밖에서도 같은 뜻**이 된다 — 선으로 나가거나 장치가 읽는 것은 그래야 한다. 그러니 필요한 자리에만 박는다.

6.2.16 명시적 변환 — `cast`

1 **값을 다른 타입으로 읽는 일은 소스에 적힌 곳에서만 일어난다.** 처리기가 스스로 하는 변환은 넓히기(§6.2.5 (2)) 하나이며, 그 밖의 변환은 모두 저자가 낱말로 적는다. 그 낱말의 중심이 `cast <타입> <값>` 이다 — **`cast` 는 이 언어에서 명시적 변환을 맡는 낱말이다.** `widen` 은 **값이 변하지 않는** 자리에만 쓰고, 값이 변할 수 있는 변환은 `cast` 로 적는다 — 부호를 바꾸거나, 갈래를 넘거나, 좁히는 자리다.

1a 변환을 적는 낱말은 셋이며 각자 맡은 자리가 다르다.

표 24 — 변환을 적는 세 낱말

낱말	맡는 자리	값이 안 맞으면
`widen`	값이 **그대로** 들어가는 넓히기(⟦§6.2.5⟧ (4) 의 표). 처리기가 스스로도 한다 — 적는 것은 눈에 보이게 하려는 것이다	일어나지 않는다
`narrow`	같은 갈래 안에서 폭을 줄인다(정수 → 더 좁은 정수)	트랩한다
`cast`	**그 밖의 모든 변환** — 부호를 바꾼다 · 정수와 부동소수점 사이 · 폭이 구현에 달린 `usize` · `idiff` 와 다른 정수 사이 · 참거짓 → 정수 · 열거 ↔ 정수	트랩한다

1a-1 `narrow` 가 갈래를 넘으면(정수 ↔ 부동소수점) 번역이 거부되고(`E-TYPE-KIND`), 대상이 값보다 넓으면 거부된다(`E-TYPE-WIDTH` — 그 낱말은 `widen` 이다).

1a-2 `narrow f32` 는 `f64` 값을 `f32` 로 **한 번** 반올림한다((3-1) 과 같은 반올림이다). 유한한 값이 `f32` 의 가장 큰 유한값을 넘어 반올림되면 — 그 값이 `f32` 에 들어가지 않으므로 — 실행이 멈춘다. 무한대와 NaN 은 그대로 넘어간다. 넘친 값을 같은 부호의 무한대로 받으려면 `cast f32` 를 쓴다.

1b **어느 쪽으로 읽을지 갈릴 수 있는 변환은 저절로 일어나지 아니하며 `cast` 를 거친다.** 같은 비트가 두 가지 수로 읽힐 수 있거나, 대상 기계에 따라 넓히기도 되고 좁히기도 되거나, 값의 갈래가 바뀌는 변환이 그렇다. 이 규칙은 앞으로 언어에 들어오는 타입과 변환에도 적용된다 — 새 변환이 애매하면 그 철자는 `cast` 이고, 암묵 변환을 늘리는 쪽으로 정하지 아니한다.

> **쉬운 말로**
>
> 변환이 소스에 적혀 있으면 읽는 사람은 **값이 바뀔 수 있는 자리를 모두 눈으로 찾을 수 있다.** `cast` 를 찾아 읽으면 그 프로그램에서 타입이 바뀌는 곳을 다 본 것이다. 처리기가 몰래 바꾸는 자리가 하나라도 있으면 그 약속은 깨진다. 그래서 이 언어는 편의를 위한 암묵 변환을 두지 않는다. 적는 품은 조금 들지만, 읽을 때 «여기서 무엇이 바뀌었나» 를 추측할 일이 없다.

2 `cast` 는 **값이 목표 타입에 들어가지 아니하면 트랩한다.** 조용히 잘리지 아니한다. 곧 `cast` 는 «아무 일이나 해도 된다» 는 표시가 아니라 "값이 변할 수 있는 자리임을 내가 안다" 는 표시다.

3 부동소수점 수를 정수로 바꾸면 **0 쪽으로 버린다** — `7.9` 는 `7` 이 되고 `-2.9` 는 `-2` 가 된다. 버려지는 것은 소수부뿐이며, 정수부가 목표 타입에 안 들어가면 (2) 대로 트랩한다.

> **주의 —**
>
> 소수부가 버려지는 것은 **트랩하지 아니한다.** 그러므로 `cast` 는 §6.2.5 (1) 의 "값을 잃는 암묵적 변환이 없다" 를 어기지 않는다 — 그 변환은 **암묵적이지 않기** 때문이다. 저자가 `cast` 라고 적었고, 그 낱말이 곧 «여기서 값이 변할 수 있다» 는 표시다.

3-1 **부동소수점 타입으로 가는** `cast` 는 — 정수에서든 다른 부동소수점 타입에서든 — 대상 타입에서 **가장 가까운 값**을 내고, 둘이 똑같이 가까우면 끝자리가 짝수인 쪽을 낸다. 반올림은 **한 번**이다 — 정수를 `f32` 로 바꿀 때 `f64` 를 거쳐 두 번 반올림하지 아니한다. 대상 타입의 가장 큰 유한값을 넘어 반올림되는 값은 같은 부호의 무한대가 되고, NaN 은 NaN 이다. 이 변환은 (2) 의 트랩을 하지 아니한다 — 부동

소수점 타입에는 넘친 값이 갈 무한대가 있다. `f32` 에서 `f64` 로는 값이 그대로다. `cast f32 <f64 1e300>` 은 `+∞` 이고, `cast f32 <u64 16777217>` 은 `16777216` 이다.

3a **`usize` · `idiff` 와 다른 정수 타입 사이**는 `cast` 로만 건넌다(§6.2 (3-3)). 이 변환은 구현의 폭에 따라 넓히기이기도 좁히기이기도 하므로 `cast` 가 그 철자다 — 값이 들어가면 그대로 내고, 들어가지 않으면 (2) 대로 트랩한다. 폭이 같거나 넓어지는 구현에서 이 `cast` 는 검사를 남기지 아니한다.

4 **참거짓에서 정수로**는 `cast` 로 간다 — `cast <정수 타입> <참거짓>` 은 참이면 `1`, 거짓이면 `0` 이다. 그 값은 어느 정수 타입에도 들어가므로 (2) 의 트랩은 일어나지 아니한다. 부동소수점 타입은 이 변환의 대상이 아니다(`E-TYPE-KIND`).

4a **정수에서 참거짓으로** 가는 `cast` 는 없다 — `bool` 은 `cast` 의 대상이 될 수 없다(`E-TYPE-KIND`). 무엇을 묻는지를 적는다: `ne n. 0 .` 또는 `eq n. 0 .`. 수의 어느 값이 참인지는 그 물음이 정한다.

4b 참거짓은 `widen` 의 표(§6.2.5 (4))에 없다 — `widen <정수 타입> <참거짓>` 은 거부된다(`E-WIDEN-KIND`). 참거짓을 수로 바꾸는 철자는 `cast` 하나다.

> **예제 (example) — 참거짓을 수로**
>
> **소스 코드 (source)**
>
> ```
> module ex_bool_cast .
>
> fn big input a u64 . output u8 . do
>   return cast u8 gt a. 2 . . .
> end .
>
> fn count_big input a u64 . input b u64 . output u64 . do
>   return add cast u64 gt a. 2 . . cast u64 gt b. 2 . . . .
> end .
> ```
>
> **실행 결과 (output)**
>
> ```
> big(5) = 1 · big(1) = 0 · count_big(5, 7) = 2
> ```

> **거부되는 예제 (rejected) — 수를 참거짓으로 바꾸는 cast 는 없다**
>
> **소스 코드 (source)**
>
> ```
> module ex_cast_to_bool .
> fn nonzero input n u64 . output bool . do
>   return cast bool n. . .     rem 무엇을 묻는지 적는다 — `ne n. 0 .`
> end .
> ```
>
> 진단: `E-TYPE-KIND`

5 **열거 값은 수가 아니다.** 수를 받는 자리 — 수로 선언된 이름 · 칸 · 인자 · 반환, 셈, 수와의 견줌, 첨자, 비트 연산, 넓히기 · 좁히기 — 에 열거 값을 두면 번역이 거부되고(`E-TYPE-KIND` · `E-TYPE-LET` · `E-TYPE-RETURN` · `E-TYPE-ARG` · `E-WIDEN-KIND`), 열거를 받는 자리에 수를 두는 것도 리터럴까지 거부된다. 갈래의 번호(§6.2.7 (2c))가 필요하면 `cast <정수 타입> <열거 값>` 으로 적는다 — 번호가 그 타입에 들어가지 않으면 (2) 대로 트랩한다. 같은 열거의 두 값은 `eq` · `ne` 로, 그리고 `lt` · `le` · `gt` · `ge` 로 견줄 수 있다 — 크기는 번호의 차례이고, 번호는 선언된 차례대로 오르므로 선언의 차례와 같다. 다른 열거의 값끼리는 견주지 아니한다(`E-TYPE-KIND`).

5a **수에서 열거로**는 `cast <열거> <정수>` 로 간다 — 그 번호를 가진 갈래를 낸다. 어느 갈래의 번호도 아닌 수이면 **트랩한다** (`E-VM-CAST`). 파일이나 선에서 읽은 수를 열거로 되돌리는 자리가 이것이다. 원천은 정수여야 한다(`E-TYPE-KIND`).

5b 값을 지닌 갈래(§6.2.19)가 하나라도 있는 열거는 `cast` 의 원천도 대상도 될 수 없다(`E-TYPE-KIND`). 번호 하나로는 그 값이 다 말해지지 않는다 — 어느 갈래인지는 `isa` · `match` 로 묻고, 값은 `<열거>.<갈래> …` 로 짓는다.

예제 (example) — 갈래의 번호는 cast 로 읽는다

소스 코드 (source)

```
module ex_enum_cast .

enum color do
  red .
  green .
  blue .
end .

fn code input c color . output u8 . do
  return cast u8 c. . .
end .

fn code_of_blue output u8 . do
  return code blue. . .
end .

fn same output u8 . do
  return cast u8 eq green. green. . . .
end .
```

실행 결과 (output)

```
code_of_blue() = 3 · same() = 1
```

거부되는 예제 (rejected) — 열거 값을 수로 낸다

소스 코드 (source)

```
module ex_enum_as_number .

enum color do
  red .
  green .
end .

fn code input c color . output u8 . do
  return c. .     rem 갈래의 번호는 `cast u8 c. .` 로 읽는다
end .
```

진단: `E-TYPE-RETURN`

예제 (example) — 읽은 수를 열거로 되돌리고 범위로 거른다

소스 코드 (source)

```
module ex_enum_from_number .

enum item do
  weapon_start 10 .
  sword .
  stick .
```

```
  weapon_end .
  tool_start 20 .
  hammer .
  tool_end .
end .

rem 1 = 무기 · 2 = 도구 — 같은 열거의 값은 크기로 견준다
fn kind_of input n u8 . output u8 . do
  let it item cast item n. . .
  if and gt it. weapon_start. . lt it. weapon_end. . . do return 1 . end .
  if and gt it. tool_start. . lt it. tool_end. . . do return 2 . end .
  return 0 .
end .

enum hangul do
  ga 11 .
  na .
  da 101 .
  ra .
end .

fn round_trip input n u32 . output u32 . do
  return cast u32 cast hangul n. . . .
end .
```

실행 결과 (output)

```
kind_of(12) = 1 · kind_of(21) = 2 · round_trip(102) = 102
```

거부되는 예제 (rejected) — 다른 열거의 값끼리 견준다

소스 코드 (source)

```
module ex_enum_cross_order .

enum color do
  red .
  green .
end .

enum size do
  small .
  large .
end .

fn f input c color . input s size . output bool . do
  return lt c. s. . .
end .
```

진단: `E-TYPE-KIND`

거부되는 예제 (rejected) — 열거로 셈한다

소스 코드 (source)

```
module ex_enum_arith .

enum level do
```

```
  low .
  high .
end .

fn next input l level . output u8 . do
  return add l. 1 . .
end .
```

진단: `E-TYPE-KIND`

6.2.17 글자를 담는 것 — 문자열은 타입이 아니다

1 이 언어에는 **문자열 타입이 없다.** 글자를 담는 것은 바이트의 조각(`slice u8`)이며, 그것이 전부다.
2 그러므로 **길이가 진실이다.** 문자열의 끝을 알려 주는 표식(널 바이트)은 값의 일부가 아니며, 길이를 세기 위해 바이트를 훑는 일은 일어나지 아니한다.

> **쉬운 말로**
>
> 널로 끝나는 문자열은 길이를 **값 안에 숨긴** 설계다. 그래서 길이를 알려면 훑어야 하고, 훑다가 표식이 없으면 그 너머를 읽는다 — 실제로 가장 흔한 메모리 사고가 여기서 났다. 조각은 길이를 **밖에 들고 다니므로** 그 사고가 설 자리가 없다.

3 씨와 만나는 자리에서만 널로 끝나는 꼴이 필요하다(§6.9.2). 그것은 **경계에서 짓는 모양**이지 이 언어가 지닌 타입이 아니다.
4 글자의 다룸 — 자르기·찾기·이어 붙이기·유니코드 검사 — 은 **라이브러리의 일**이다 (§9.5). 처리기는 조각과 바이트만 안다.

6.2.18 뜻이 없는 타입 이름

1 어떤 이름은 **받아들여지되 뜻이 없다.** 그런 이름을 시그니처에 적으면 처리기는 경고한다(`W-NOT-YET`). 그 이름들은 부록 A.9 (2) 에 모여 있다 — `byte` · `char` · `str` · `string` · `bytes_view` · `dyn` · `atomic` 과 같은 것들이다.
2 뜻이 없다는 것은 **처리기가 그 이름이 담는 값의 모양을 모른다**는 말이다. 크기도, 배치도, 그 값에 할 수 있는 일도 정해지지 않았으므로, 그 이름에 기대는 검사는 **서지 아니한다**.

> **주의 —**
>
> 이 자리가 **말해야 하는** 까닭이 여기에 있다. 번역은 통과하고 답도 나오므로, 서지 않은 검사는 어디에도 드러나지 않는다. **조용히 빠진 것은 아무도 못 찾는다** — 그래서 처리기가 말한다.

3 이름에 뜻을 주는 길은 둘이다. **바탕이 되는 타입을 직접 적거나**, 그 이름을 선언하는 것이다 (`type str option u64 .`). 선언한 뒤에는 뜻이 있으므로 경고하지 아니한다.

> **예제 (example) — 뜻 없는 이름은 경고를 받되 거절되지는 않는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_name_only .
>
> rem `str` 은 내장이 아니다 — 뜻을 주지 않으면 W-NOT-YET 을 받는다. 여기서는 선언하여 뜻을 주었다.
> type str option u64 .
>
> fn f input s str . output u8 . do return 1 . end .
> ```
>
> **실행 결과 (output)**
>
> ```
> f() = 1
> ```

6.2.19 값을 지닌 갈래 — 짓기와 해체

1 `enum` 의 갈래는 값을 지닐 수 있다. 갈래를 적을 때 지닐 것의 이름과 타입을 함께 적는다.

2 그런 값을 짓는 모양은 `<열거>.<갈래>` 이며, 갈래가 지니기로 한 것을 **모두** 준다. 모자라면 거부된다(`E-ENUM-ARITY`) — 조용히 반만 지은 값을 만들지 아니한다.

3 값을 지닌 열거를 다루는 길은 둘이다.

표 25 — 값을 지닌 갈래를 다루는 두 op

op	모양	무엇을 하나
`isa`	`isa <값> <갈래>`	지금 그 갈래인가를 묻는다
`payload`	`payload <값> <갈래> <이름>`	그 갈래가 지닌 것을 읽는다

4 없는 갈래를 부르면 거부되고(`E-ENUM-NOVARIANT`), 그 갈래가 지니지 않는 이름을 읽으려 하면 거부된다(`E-ENUM-NOFIELD`). 둘 다 **번역할 때** 판정된다 — 갈래와 이름은 소스에 적혀 있으므로 실행을 기다릴 까닭이 없다.

5 갈래가 여럿인 열거에서 `payload` 는 **먼저 그 갈래임이 좁혀진 뒤에만** 쓸 수 있다 (`E-ENUM-UNCHECKED`). 좁히는 것은 `isa` 이며, 보통 `guard isa <값> <갈래> else …` 로 적는다.

5a 이미 **다른 갈래**로 좁혀진 자리에서 `payload` 을 쓰는 것도 거부된다 (`E-ENUM-VARIANT`) — 그 자리에서는 다른 갈래임이 이미 증명되었으므로, 그 `payload` 는 **틀린 것을 읽는다.**

5b 갈래가 지니는 것의 타입은 **낱말 하나로 적는다**(`E-ENUM-PAYLOAD`). 갈래 한 줄에는 칸 사이를 가르는 표시가 없어서, 여러 낱말로 적는 타입(`slice u8 · option t · owned t · ref t`)은 어디서 끊기는지 정해지지 아니하기 때문이다. 이것은 철자의 제한이지 크기의 제한이 아니다 — 지니는 값의 크기는 묻지 아니한다. 여러 낱말 타입은 그것을 칸으로 가진 구조체에 이름을 붙여 그 이름으로 적는다.

> **쉬운 말로**
>
> (5) 가 하는 일은 「없는 것을 읽는 일」을 막는 것이다. 좁히지 않고 읽으면 그 값이 그 갈래일 때만 맞고 아닐 때는 **엉뚱한 바이트**를 읽는다 — 그것은 답이 틀리는 것이 아니라 답이 **어쩌다 맞는** 것이며, 이 언어가 가장 싫어하는 모양이다.

> **예제 (example) — 여러 낱말 타입은 이름 붙인 구조체로 지닌다**
>
> **소스 코드 (source)**
>
> ```
> module ex_enum_named_payload .
>
> struct bytes do
>   b slice u8 .
> end .
>
> struct pt do
>   x u64 .
>   y u64 .
>   z u64 .
> end .
>
> enum msg do
>   data d bytes .
>   dot p pt .
>   blank .
> end .
>
> fn size_of_data input m msg . output usize . do
> ```

```
  guard isa m. data . else return 0 . .
  let d bytes payload m. data d . .
  return len field d. b . . .
end .

export fn f input s slice u8 . output usize . do
  let m msg msg.data lit bytes do b s. . end . . .
  return size_of_data m. . .
end .

export fn g output u64 . do
  let m msg msg.dot lit pt do x 1 . y 2 . z 7 . end . . .
  guard isa m. dot . else return 0 . .
  let p pt payload m. dot p . .
  return add add field p. x . field p. y . . field p. z . . .
end .
```

실행 결과 (output)

```
f([1,2,3]) = 3 · g() = 10
```

6 열거가 자기 자신을 값으로 지니면 크기가 정해지지 아니하므로 거부된다 (E-ENUM-INFINITE). 되풀이되는 구조는 **사이에 무언가를 두어** 짓는다.

> **쉬운 말로**
>
> 값을 지닌 갈래는 새 짜임이 아니다. 갈래를 가리키는 칸 하나와 갈래가 지닌 것을 담은 묶음이 있을 뿐이며, 짓는 것은 묶음 짓기이고 읽는 것은 칸 읽기다. 그래서 이 기능은 **새 기계 명령을 하나도 늘리지 아니했다** — 이미 있던 것으로 낮아지므로 두 뒤끝 모두에서 저절로 돌았다.

6.2.20 흩어진 조각을 하나로 보기 — segments

1 이어져 있지 않은 여러 조각을 **하나의 눈**으로 볼 수 있다. 뒤에 놓인 바이트와, 그 안에서 어디부터 얼마인지를 적은 서술자 둘로 이루어진다.

표 26 — 흩어진 조각을 다루는 op

op	모양	무엇을 하나
view_segments	view_segments <바이트> <서술자>	둘을 묶어 하나의 눈을 만든다
segview.piece_count	segview.piece_count <눈>	조각이 몇인지 센다(표준 모듈 segview)
segview.piece_at	segview.piece_at <눈> <번호>	그 번호의 조각을 조각(slice)으로 준다(표준 모듈 segview)

1a 눈은 칸 back(뒤의 바이트)과 descs(서술자)를 든 묶음이다. 조각의 수와 번호의 조각은 그 칸을 읽고 잘라내는 것으로 쓰이므로 언어의 연산이 아니라 표준 모듈 segview 의 op 이다(§9.3).

2 서술자는 「어디부터」와 「얼마」의 짝을 담은 조각이다. 그러므로 이 셋은 **새 타입도 새 기계 명령도 늘리지 아니한다** — 묶음 짓기와 칸 읽기와 잘라내기로 낮아진다.

> **쉬운 말로**
>
> 흩어진 조각을 다루는 길이 없으면, 쓰는 사람은 그것을 **한 자리에 모으는** 수밖에 없다. 모으는 일은 베끼는 일이고, 베끼지 않아도 되는 곳에서 베끼는 것이 이 언어가 없애려는 비용이다. 눈을 주면 베낌이 사라진다.

6.2.21 미리 끌어오기 — `prefetch`

prefetch <조각> <번호> 는 그 자리의 바이트를 **곧 쓸 것**이라고 기계에 알린다.
이것은 **힌트일 뿐**이다. 값을 남기지 아니하고, 프로그램의 뜻을 바꾸지 아니하며, 기계가 무시해도 답은 같다.

> **주의 —**
>
> 힌트는 **틀려도 프로그램이 틀리지 않는** 것만 힌트다. 뜻을 바꾸는 것은 힌트가 아니라 명령이며, 그런 것은 이 자리에 오지 아니한다.

6.2.22 타입이 맞는다는 것

값을 두는 모든 자리에서 값의 타입은 그 자리가 적은 타입과 **같아야** 한다.

표 27 — 타입이 어긋나는 자리

진단	어디에서
E-TYPE-LET	let 으로 묶을 때 — 주는 값이 적은 타입과 다르다
E-TYPE-VAR	var 로 묶을 때 — 같은 어긋남이되 자리가 다르다
E-TYPE-SET	고쳐 쓸 때 — 새 값이 그 이름의 타입과 다르다
E-TYPE-RETURN	돌려줄 때 — 돌려주는 것이 op 이 적은 것과 다르다
E-TYPE-ARG	건넬 때 — 인자가 매개변수와 다르다
E-TYPE-FIELD	**만들 때** — 없는 칸을 주거나, 칸의 값이 적은 타입과 다르거나, 칸을 빠뜨렸다
E-TYPE-TRY	try 할 때 — 벗겨 낼 것이 없는 값이다

묶음과 열거는 struct <이름> do … end . · enum <이름> do … end . 로 선언한다. 다른 모양으로 선언하려 하면 거부된다(E-TYPE-DECL).
칸의 배치가 같아도 **이름이 다르면 다른 묶음**이다(E-TYPE-STRUCT, §8.9).

> **쉬운 말로**
>
> 이 표의 일곱은 모두 같은 문장의 다른 자리다 — "여기 오는 것의 타입은 이것이다"를 소스가 적었고, 처리기는 그 적힌 것과 실제를 견준다. 자리마다 진단을 따로 두는 까닭은 **어디에서 어긋났는지**가 고치는 사람에게 가장 필요한 사실이기 때문이다.

6.2.23 칸을 적는 꼴

묶음의 칸은 **이름 다음에 타입**이다(x u64 .). 그 꼴이 아니면 거부된다 (E-FIELD-FORM).
칸을 읽을 때는 field <값> <이름> … 으로 적는다. 점을 붙여 읽는 옛 철자는 이 언어에 없다(E-FIELD-GLUED).
vec 과 가림막은 **값 그 자체**이며, 가리키거나 빌리는 표지를 붙일 수 없다 (E-VEC-QUAL). 그 레인들은 낱말 안의 자리이지 메모리의 자리가 아니다.

> **거부되는 예제 (rejected) — 칸은 이름 다음에 타입이다**
>
> **소스 코드 (source)**
>
> ```
> module ex_field_form .
>
> struct p do
>   x .              rem 타입이 없다
> end .
> ```

```
fn f output u8 . do return 1 . end .
```

진단: `E-FIELD-FORM`

6.2.24 임의 폭 정수 — `bits`

`type <이름> bits <수> .` 은 그 수만큼의 비트를 가진 정수 타입을 만든다. 폭은 **1 부터 64 까지**다. 폭은 2 의 거듭제곱일 필요가 없다 — `bits 3` 도 `bits 10` 도 선다.

> **주의 —**
>
> 폭이 기계의 낱말과 어긋나면 **빠름은 보장되지 아니한다.** 이 언어가 보장하는 것은 **정확함**이다 — `bits 10` 은 열 비트로 셈한 답을 주되, 그것이 열여섯 비트 셈보다 빠르다고 말하지 아니한다. 비용이 보이지 않는 자리를 만들지 않으려면, 보장하지 않는 것을 **보장하지 않는다고 적어야** 한다.

6.2.25 정했으되 짓지 않은 타입 낱말 — `c8…c64` · `b8…b128` · `list` · 겹치는 합 · 자식 영역

이 조항의 낱말은 §6.9.4 의 넷처럼 **뜻이 정해져 있되 처리기가 아직 짓지 않은** 것이다. 철자가 정해진 셋(아래 (2) (4))은 타입 자리에 쓰면 번역이 거부된다(`E-TYPE-UNBUILT`) — 받아서 뜻 없는 이름으로 두거나 다른 타입으로 돌리지 아니한다. 셋 모두 예약어가 아니다: 저자가 그 이름의 타입을 선언하면 (`type c8 u8 .`) 그 선언이 서고, 같은 이름의 지역 · op 도 선다.

`c8` · `c16` · `c32` · `c64` 는 **글자의 부호 단위**(code unit)다 — 그 수만큼의 비트로 된, 어떤 인코딩의 한 단위. 수가 아니므로 셈하지 아니한다. 정수와 오가는 것은 `cast` 로만이다(§6.2.16).

`b8` · `b16` · `b32` · `b64` · `b128` 은 **속을 모르는 바이트 덩이**다 — 그 수만큼의 바이트이고, 정렬은 크기다. 셈도 견줌도 없이 옮기고 베끼기만 한다. 수나 구조로 읽으려면 창(§6.2.13)을 거친다.

`list` 는 **원소마다 타입이 다를 수 있는 목록**이다. 원소마다 그 타입의 표지와 값의 자리를 지니며, 핸들로만 닿는다(값으로 놓이는 확정 타입이 아니다). 그 목록의 원소를 쓰는 op 은 표지로 갈래를 고른다.

겹치는 합 — 같은 자리에 갈래 가운데 하나를 놓는 타입이며, 크기는 가장 큰 갈래의 것이다. 철자는 짓을 때 정한다: 원안의 `union` 은 언어가 짧은 낱말을 저자에게 남긴다는 규칙(§6.3.3 (1e) — 집합의 합도 `bitset_union` 이다)과 부딪친다. 그래서 이 갈래는 낱말로 거부되지 않는다.

자식 영역 — 다른 영역에서 깎아 낸 영역이며, 부모가 끝날 때 함께 끝난다(어휘 블록이 끝날 때가 아니다). 깎아 내는 것은 새 메모리를 얻지 않으므로 `alloc` 효과가 아니다(§8.13). 원안의 철자 (`child_of`)는 지금 문법에 자리가 없으므로, 철자는 짓을 때 정한다.

유예. 이 낱말들은 쓰임이 나올 때 짓는다. 다른 길이 그 쓰임을 다 채워 **필요 없다고 보이면 버린다.** 지금 그 쓰임을 맡은 것: 부호 단위는 정수(`u8` · `u16` · `u32`)와 라이브러리 `utf8` · `utf16` · 바이트 덩이는 `slice u8` · `array N u8` 과 창 · `list` 는 값을 담는 열거(§6.2.7) · 겹치는 합은 값을 담는 열거 · 자식 영역은 얼로케이터(§8.13). 짓을지 버릴지는 소유자가 정한다. 그때까지 이 조항은 남고 처리기는 (2) (4) 의 낱말을 계속 알아본다.

> **거부되는 예제 (rejected) — 아직 짓지 않은 타입 낱말**

소스 코드 (source)

```
module ex_unbuilt_type .

fn f input x c8 . output u64 . do
  return 0 .
end .
```

진단: `E-TYPE-UNBUILT`

참고 (informative)

출처는 RFC-0002 §T3 · §S · §LIST(부호 단위 · 바이트 덩이 · 겹치는 합 · 이질 목록)와 RFC-0004(자식 영역)다. 두 RFC 는 채택된 뒤 보관됐는데, 이 낱말들은 정본으로 옮겨지지 않았다(2026-11-11 훑기 F5). 소유자는 §6.9.4 와 같이 «정식으로 넣되, 차후 불필요하게 되면 버린다고 명시해서 계속 추적» 하기로 했다(2026-11-12). RFC-0002 스스로 «`union` 이 정말로 필요한 자리가 나오는지(지금은 값을 담는 열거가 그 자리를 덮는다)» 를 남은 물음으로 적었다.

6.3 식과 연산 (Expressions and operations)

1 식 (expression) 은 값을 만들어 내는 것이다. 리터럴도 식이고, 이름도 식이며, 연산을 적용한 것도 식이다.

6.3.1 전위 표기가 기본이다

1 식은 전위 표기 (prefix notation) 로 적는다 — 연산의 이름이 먼저 오고 인자가 뒤에 온다.

2 전위 표기에는 **우선순위 규칙이 없다.** 어떤 것이 먼저 계산되는지는 점과 순서가 전부 말해 준다 — 폼은 이름으로 열리고 제 점으로 닫히며(§6.1.5), 안쪽 폼이 먼저 닫힌다.

2a 부름을 겹쳐 적는 데 괄호가 들지 아니한다 — `add 1 mul a. b. . .` 에서 첫 점이 `mul` 을, 둘째 점이 `add` 를 닫는다. 폼의 끝은 **그 폼의 점**이 정한다. 연산이 받는 인자 수는 끝을 정하지 아니한다 — 인자 수가 맞는지는 폼이 닫힌 뒤에 따로 본다(맞지 아니하면 `E-IR-ARITY`).

2b 괄호는 **묶음을 눈에 띄게 하는 꾸밈**이다 — `add 1 (mul a. b. .) .` 는 (2a) 의 식과 같은 식이다. 괄호 안의 폼도 제 점으로 닫히고, 괄호는 뜻을 바꾸지 아니한다.

2c **피연산자를 계산하는 차례는 자유 선택 (free choice) 이다**(§4.3). 처리기는 한 연산의 피연산자들을 어느 차례로든 계산할 수 있다. 기본 연산 · 빌트인 연산과 저자가 선언한 op 이 같다. 다만 다음은 정해져 있다.

- 피연산자를 **모두 계산한 뒤에** 그 연산을 한다.
- 피연산자마다 **한 번씩만** 계산한다.
- **효과를 내는 피연산자들은 적힌 차례로, 왼쪽부터 계산한다.** 여기서 효과는 `panic` 밖의 효과를 말한다(§7) —

멈출 수 있다는 것만으로는 차례가 묶이지 아니한다.

- `and` 와 `or` 는 예외다 — 앞 피연산자를 먼저 계산하고, 그것만으로 결과가 정해지면 뒤 피연산자를 계산하지 아니한다(§6.3.3 (4)).

자유 선택 (free choice) — FC-0001 — 피연산자를 계산하는 차례

처리기는 한 연산의 피연산자들 가운데 **효과를 내지 않는 것들**을 어느 차례로든 계산할 수 있다. 고를 수 있는 범위는 그 차례뿐이다 — 피연산자를 모두 계산한 뒤에 연산하는 것, 하나를 한 번씩만 계산하는 것, 효과를 내는 피연산자들이 적힌 차례로 계산되는 것은 정해져 있다. 대부분의 프로그램에서는 차례가 달라도 답이 같다. 차이가 드러나는 때는 하나다 — 두 피연산자가 서로 다른 까닭으로 멈출 때 **어느 멈춤이 나는가**. 그 차이가 중요하면 먼저 계산할 것을 `let` 으로 이름에 담는다 — 문장은 적힌 차례로 실행된다.

예제 (example) — 전위 표기

소스 코드 (source)

```
let a u32 add 1 2 . .
let b u32 add 1 mul 2 3 . . .
let c bool lt a. b. . .
```

> **쉬운 말로**
>
> `1 + 2 * 3` 을 처음 보는 사람은 `*` 가 먼저인지 `+` 가 먼저인지 **외워야** 안다. `add 1 mul 2 3 . .` 은 외울 것이 없다 — 점이 그대로 말해 준다. 이 언어가 전위를 기본으로 삼는 이유가 그것이다.

6.3.2 중위 표기 — `expr` 섬

1 산술이 길어지면 읽기 어려우므로, `expr` 로 시작하는 자리에서는 중위 표기 (infix notation) 를 쓸 수 있다. 이 자리를 섬 (island) 이라 한다.

2 섬 안의 표기는 **전위와 완전히 같은 뜻**으로 바뀐다. 섬은 표기의 투영일 뿐이며 실행 비용이 없다.

3 우선순위는 **오직 섬 안에만** 있다. 다음 표가 그 전부다.

표 28 — `expr` 섬 안의 우선순위 (강한 것이 위)

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)

4 `and` 가 `or` 보다 강하게 묶는다. 곧 `expr a or b and c` 는 `or a. and b. c. . .` 와 같다.

5 비교는 **이어 쓸 수 없다.** `expr a lt b lt c` 는 거부된다.

6 위 표가 섬의 **전부**다. 이 집합은 닫혀 있으며 늘어나지 않는다. 표에 없는 연산은 — 비트 연산 (§6.3.5) · 나머지 `mod` · 최소 `min` 최대 `max` 를 비롯해 — 섬 안에서도 **전위로 적는다.**

7 단항 연산도 섬에 없다. 부호를 뒤집으려면 `expr 0 - a` 로 적거나 전위 폼을 괄호로 묶는다(`expr (not b. .) or c. .`).

> **주의 — 섬이 작은 것은 부족해서가 아니다**
>
> 중위 표기의 값은 **읽기 쉬움** 하나뿐이고, 그것은 우선순위를 **아무도 찾아보지 않아도 될 때만** 성립한다. 표의 열둘은 학교에서 배운 셈과 같아 모두가 같게 읽는다 — 곱셈이 덧셈보다 강하게 묶는다는 데 이견이 없다. 비트 연산은 정확히 그 반대다. 씨(C)에서 `a & b == c` 는 `(a & b) == c` 가 아니라 `a & (b == c)` 로 묶인다. $a \cdot b \cdot c$ 가 모두 6 일 때 앞의 것은 참이고 뒤의 것은 거짓이다 — 답이 갈린다. 이것은 씨를 오래 쓴 사람도 걸리는 함정이며, 여러 언어가 이 우선순위를 서로 다르게 정해 두었다. 곧 비트 연산을 섬에 넣으면 **읽기 쉬워지는 것이 아니라 찾아봐야 하는 것이 늘어난다.** 전위 표기에는 우선순위가 아예 없으므로(§6.3.1) 괄호가 답을 말한다 — 길어지지만 **틀릴 수 없다.** 이 언어는 짧은 쪽이 아니라 틀릴 수 없는 쪽을 고른다 (§1.3).

> **쉬운 말로**
>
> 표기 투영은 읽기 쉬우라고 있는 것이지, 모든 연산에 **두 번째 문법**을 주려고 있는 것이 아니다. 같은 뜻을 적는 길이 둘이면 읽는 사람이 둘 다 알아야 한다 — 그 값은 길이를 줄여서 얻는 것보다 크다.

> **예제 (example) — 중위 표기**
>
> **소스 코드 (source)**

```
module ex_infix .

export 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 .
```

거부되는 예제 (rejected) — 비교를 이어 쓸 수 없다

소스 코드 (source)

```
module ex_chain .

proc p input a u32 . input b u32 . input c u32 . output bool . effects none .
do
  return expr a. lt b. lt c. . .     rem 두 비교를 이어 쓸 수 없다
end .
```

진단: `E-EXPR-CHAIN`

주의 — 비교를 이어 쓰지 못하는 이유

수학에서 `a < b < c` 는 「a 가 b 보다 작고 b 가 c 보다 작다」는 뜻이다. 그런데 여러 언어에서 그 표기는 **다른 뜻**으로 읽힌다(먼저 `a < b` 를 계산해 참거짓을 얻고, 그것을 다시 `c` 와 비교한다). 같은 글자가 사람과 기계에게 다른 뜻이면 그것은 조용히 틀리는 자리다. 그래서 이 언어는 아예 거부한다.

6.3.3 기본 연산과 빌트인 연산

1 언어가 뜻을 정해 둔 연산은 두 갈래다. 기본 연산 (core op) 은 어디서나 제 이름만으로 부르며, 저자가 지은 이름과 **한 이름공간**을 쓴다. 빌트인 연산 (builtin op) 은 `call_builtin` 뒤에서만 부르며, **제 이름공간을 따로** 갖는다((1c) · (1f)). 두 갈래 모두 이름의 목록은 고정되어 있고, 전체는 부록 D 에 있다. 부록 D 는 연산마다 **무엇을 받고 무엇을 내는지, 언제 번역이 거부되고 언제 실행이 멈추는지**도 적는다.

1a 기본 연산의 이름이 모두 같은 규칙을 갖지는 아니한다. 대부분은 지역 이름으로 쓸 수 없으나(§6.1.3), 일부는 **문맥 안에서만** 연산이므로 그 밖에서는 저자의 이름이 될 수 있다. 어느 쪽인지는 부록 D 가 적는다.

1b 아래에 드는 것은 **기본이 되는 것을 갈래별로 보인 것**이지 전부가 아니다. 전체 목록과 각 연산의 뜻은 부록 D 에 있다.

1c 빌트인 연산은 머리말 `call_builtin` 뒤에서만 선다: `call_builtin <이름> <피연산자>*`. 그 이름은 **닫힌 집합**이며 부르는 자리는 그 하나뿐이다. 어느 이름이 그러한지는 부록 D 가 적는다. 바깥 세상에 닿는 것 — 파일 · 그물 · 프로세스(`file_*` · `dir_*` · `path_remove` · `path_rename` · `link_type` · `net_*` · `proc_*`), 표준 입출력(`write_out` · `read_in`), 인자와 환경(`arg` · `env_get`), 시계(`time_now` · `time_local` · `time_sleep`), 터미널(`tty_raw` · `tty_read` · `tty_size`), 난수(`random_bytes`), reactor(`reactor_new` · `r_read` · `r_write`) — 은 첫 피연산자가 권한이다: `call_builtin net_send <권한> <핸들> <바이트>`.

1c-1 권한을 받지 아니하는 빌트인 연산은 **계산**이다 — 부동소수점 함수(§6.3.9), 조각의 합(§6.3.7.1), 그리고 해시 · 암호 · 캐리 없는 곱셈 · 난수 열(`hash_bytes` · `crc32` · `sha256` · `sha384` · `sha512` · `aes_round` · `aes_round_last` · `aes_ctr` · `ghash` · `aes_gcm` · `chacha20` · `poly1305` · `chacha_poly` · `clmul_lo` · `clmul_hi` · `mul_hi` · `rng_next`).

1c-2 표준 라이브러리는 이 빌트인 연산들을 감싼 op 을 내놓는다(§9.3 (1a)) — `stdio` · `args` · `env` · `clock` · `tty` · `random` · `reactor` · `math` 따위다. 감싸개는 피연산자를 그대로 넘기므로 뜻이 같다.

1f **빌트인 연산의 이름공간은 저자의 이름공간과 따로다.** `call_builtin` 바로 뒤의 낱말은 언제나 빌트인 연산의 이름으로 읽고, 그 밖의 자리에 선 같은 철자는 언제나 저자의 이름으로 읽는다. 그러므로 저자는 빌트인 연산과 같은 철자를 제 op · 매개변수 · 지역의 이름으로 지을 수 있고, 한 단위 안에서 둘을 함께 쓸 수 있다. 낱말이 선 자리가 어느 쪽인지를 정하므로, 한 철자가 한 자리에서 두 뜻을 갖는 일은 없다.

> **쉬운 말로**
>
> 빌트인 연산은 **보통의 프로그램이 쓰라고 둔 것이 아니다.** 라이브러리로는 적을 수 없는 일 — 기계의 명령으로 내려가는 계산, 바깥 세상에 닿는 일, 처리기만이 아는 일 — 을 처리기가 가려서 내놓는 문이다. 이 문을 쓰는 것은 시스템 라이브러리를 짓는 사람과 처리기를 짓는 사람이고, 응용 프로그램은 그것을 감싼 표준 라이브러리의 op 을 부른다(§9). 이름공간을 따로 둔 까닭도 여기 있다 — 라이브러리가 `sha256` 같은 자연스러운 이름으로 감싸개를 내놓을 수 있어야 한다.

> **쉬운 말로**
>
> 왜 자리에 가두는가. 한 프로그램이 **한 번** 쓰는 연산이 모두의 어휘를 늘리면, 읽는 사람이 외울 이름이 는다. 그 비용은 쓰는 사람 하나가 아니라 **읽는 사람 전부**가 치른다. 같은 까닭으로 파이프의 스테이지 이름도 `pipe` 안에서만 뜻이 있고(§6.12), 타입 낱말도 제 자리에서만 선다.

1d 맨 이름으로 부른 낱말이 저자의 이름 가운데 없고 빌트인 연산의 철자와 같으면, 처리기는 그것을 `call_builtin` 을 빠뜨린 것으로 보고 거부한다(`E-BUILTIN-BARE`). `call_builtin` 뒤에 그 닫힌 집합 밖의 낱말을 놓는 것도 적합하지 아니하다(`E-BUILTIN-NAME`) — 그 자리는 저자의 이름으로 조용히 넘어가지 아니한다.

> **거부되는 예제 (rejected) — 빌트인 연산을 맨 이름으로 부를 수 없다**
>
> **소스 코드 (source)**
>
> ```
> module ex_bare .
>
> proc p input s slice u8 . input o mut slice u8 . output u64 . effects none .
> do
>   return sha256 s. o. . .
> end .
> ```
>
> 진단: `E-BUILTIN-BARE`

> **예제 (example) — 같은 것을 자리에 맞게 적으면 선다**
>
> **소스 코드 (source)**
>
> ```
> module ex_call .
>
> proc p input s slice u8 . input o mut slice u8 . output usize . effects none .
> do
>   return call_builtin sha256 s. o. . .
> end .
> ```

> **예제 (example) — 저자의 `sha256` 과 빌트인 연산 `sha256` 이 한 단위에 함께 선다**
>
> **소스 코드 (source)**
>
> ```
> module ex_both .
>
> fn sha256 input x u64 . output u64 . do
> ```

```
  return add x. 1 . .
end .

proc p input s slice u8 . input o mut slice u8 . output u64 . effects none .
do
  let n usize call_builtin sha256 s. o. . .
  return sha256 cast u64 n. . . .
end .
```

1e 언어가 뜻을 정한 이름을 전역 어휘에서 가두는 길은 셋이다.

- **머리** — `call_builtin <이름>` 의 이름 자리((1c)). 빌트인 연산이 그렇다.
- **접두사** — 한 무리의 op 이 같은 앞마디를 나눈다: `bitset_*`(작은 수의 집합, §6.2.12) · `lane_*` 와 `reduce_*`(레인) · `atomic_*` · `wrap_*` · `sat_*` · `chk_*` · `narrow_*`. 접두사가 붙은 이름은 전역 이름이지만, 그 무리가 짧은 낱말(`union` · `remove` · `select` · `any` …)을 차지하지 아니하므로 저자가 그 낱말을 제 이름으로 쓸 수 있다.
- **자리** — 정해진 문맥 안에서만 뜻이 있는 낱말. `pipe` 의 스테이지와 종결자가 그렇다(§6.12): 종결자 `any` · `all` 은 `pipe` 안의 낱말이고, 레인의 판정은 `lane_any` · `lane_all` 이다. 한 op 의 철자는 하나다. 이름을 옮길 때 옛 철자를 동의어로 남기지 아니한다.

2 산술 연산은 `add sub mul div mod` 이다. 나머지를 내는 연산의 철자는 `mod` 이며, `rem` 은 **주석을 여는 낱말**이다(§6.1.2) — 같은 철자를 두 뜻에 쓰지 아니한다.

2a `mod` 는 **정수의** 나머지다. 부동소수점 수의 나머지는 `fmod`(§6.3.9)이며, 부동소수점 피연산자에 `mod` 를 쓰면 번역이 거부된다(`E-TYPE-KIND`, 2026-11-02 — 그 전에는 검사를 지나 실행 중에 멈췄다).

3 비교 연산은 `eq ne lt le gt ge` 이며 결과는 `bool` 이다.

3a 정수끼리의 비교는 산술과 같은 규칙을 따른다 — 두 피연산자가 넓히기로 견줄 수 있어야 하고, 그럴 수 없으면 번역이 거부된다(`E-TYPE-SIGN`, §6.3.4 (1)). 처리기는 비교를 위해 부호를 몰래 바꾸지 아니한다. 부호나 폭이 달라도 두 정수의 **수학적 값**을 견주려면 표준 라이브러리 `intops` 의 `cmp_eq` · `cmp_lt` · `cmp_le` 를 이름으로 부른다(§9) — `intops.cmp_lt i64. u64. a. b. .` 는 `a` 가 -1 이고 `b` 가 0 일 때 참이다. 그 op 들은 어느 정수 타입의 짝이든 받는다(`usize` · `idiff` · 128 비트 포함).

4 논리 연산은 `and or not` 이며 피연산자와 결과가 모두 `bool` 이다. `and` 와 `or` 는 단락 평가 (short-circuit) 한다 — 앞쪽만으로 결과가 정해지면 뒤쪽을 계산하지 않는다.

5 슬라이스 연산은 `len idx` 이다(§6.2.6).

6 변환 연산은 `widen narrow` 이다(§6.2.5).

7 비트 연산은 §6.3.5 에 있다.

자유 선택 (free choice) — FC-0002 — min 과 max 에 NaN 이 끼면

> 부동소수점 수의 `min` · `max` 에서 피연산자 하나가 NaN 이면, 처리기는 NaN 이 아닌 쪽의 값을 내거나 NaN 을 낸다. 둘 가운데 어느 것인지는 처리기가 고른다. 두 피연산자가 모두 NaN 이 아니면 결과는 하나로 정해져 있다. 기계의 최소 · 최대 명령이 이 자리에서 서로 다르게 답하므로, 한쪽으로 못 박으면 명령 하나로 끝날 일이 느려진다.

8 `neg` 는 부호를 뒤집는다. 피연산자는 **부호 있는 정수이거나 부동소수점 수**여야 한다 — 부호 없는 정수에는 음수가 없으므로 번역이 거부된다(`E-TYPE-SIGN`). 결과의 타입은 피연산자의 타입이다.

8a `neg` 도 산술이다. 결과가 그 폭에 들어가지 않으면 멈춘다(§6.3.4 (3)) — 부호 있는 타입의 가장 작은 값이 그 경우다(`i8` 의 -128).

쉬운 말로

이름이 낯설어 보이지만 규칙은 하나다 — **기호 대신 낱말**이다. `+` 는 `add`, `<` 는 `lt`(less than), `==` 는 `eq`(equal) 다. 기호를 안 쓰는 이유는 두 가지다: 기호는 언어마다 뜻이 미묘하게 다르고(예: `^` 는 어디선 거듭제곱, 어디선 배타적 논리합), 말로 읽을 수 있는 표기가 음성 입력이나 낭독에도 그대로 쓰이기 때문이다.

6.3.4 산술의 규칙

1 이항 산술의 두 피연산자는 **넓히기 관계로 비교 가능**해야 한다(§6.2.5). 비교할 수 없으면 번역이 거부된다. 폭이 다르기만 하면 넓은 쪽으로 맞추어 계산한다.

1a 부호가 섞여도, **값을 지키는 넓히기가 있으면** 계산할 수 있다. 부호 없는 쪽이 결과 타입 안에 값을 잃지 않고 들어가면 된다 — `u8` 과 `i16` 은 섞을 수 있다.

1b 그런 넓히기가 없으면 번역이 거부된다. `u32` 와 `i32` 는 폭이 같아 부호 없는 쪽이 부호 있는 쪽에 들어가지 못하므로 섞을 수 없다.

2 결과의 타입은 두 피연산자 중 **넓은 쪽**이다. 처리기는 두 피연산자 어느 것도 아닌 제3 의 타입을 만들어 내지 아니한다.

3 산술은 **선언된 폭에서** 일어난다. 결과가 그 폭에 들어가지 않으면 **트랩한다**.

4 정수를 0 으로 나누면 트랩한다. 부동소수점 수를 0 으로 나누면 IEEE 754 가 정한 대로 무한이 된다.

5 `div` 와 `mod` 는 **선언된 부호**로 해석한다.

5a `div` 는 몫을 **0 쪽으로 자른다** — `div -7 2 .` 는 −3 이다.

5b `mod` 는 **바닥 나눗셈의 나머지**다. 나머지의 부호는 **제수**를 따른다 — `mod -7 2 .` 는 1, `mod 7 -2 .` 는 −1 이다. 그러므로 제수가 양수이면 결과는 언제나 0 이상이고 제수보다 작다: `0 < n` 이면 `0 ≤ mod i n < n` 이며, `i` 가 음수여도 그렇다.

5c 그러므로 `div` 와 `mod` 는 **서로 짝이 아니다.** `(div a b) × b + (mod a b)` 가 `a` 와 같은 것은 두 수의 부호가 같거나 나누어떨어질 때뿐이다. 0 쪽으로 자른 몫과 짝인 나머지는 `sub a. mul div a. b. . b. . .` 로 적는다.

5d 부호 있는 타입의 가장 작은 값을 −1 로 나눈 `mod` 는 0 이다. `div` 는 같은 자리에서 멈추지만(몫이 그 폭에 없다) 나머지 0 은 그 폭 안에 있다.

5e 짝이 맞는 몫과 나머지는 표준 라이브러리 `intops` 에 있다(§9) — `div_floor` 는 `mod` 의 짝인 **바닥 몫**이고 (`(div_floor a b) × b + (mod a b)` 는 언제나 `a` 다), `mod_trunc` 는 `div` 의 짝인 **자른 나머지**다(부호는 피제수를 따른다 — `mod_trunc -7 2` 는 −1). 둘 다 타입을 인자로 받고(`intops.div_floor i64. a. b. .`), 0 으로 나누면 멈추며, 부호 있는 타입의 가장 작은 값을 −1 로 나눈 `div_floor` 도 `div` 처럼 멈춘다. 모두 로우엔트로 적혀 있다(§9.3 (5)).

쉬운 말로

왜 `mod` 가 제수의 부호를 따르는가. 나머지를 쓰는 자리는 거의 언제나 **자리를 고르는 일**이다 — 원형 버퍼의 칸, 해시 표의 칸, 요일. 그 자리에서 필요한 성질은 «결과가 0 이상이고 크기보다 작다» 이고, 바닥 나눗셈의 나머지만이 음수에서도 그것을 지킨다. C 의 `%` 는 `-7 % 2` 를 −1 로 내므로 그 성질이 깨진다.

예제 (example) — 부호가 섞여도 값을 지키는 넓히기가 있으면 된다

소스 코드 (source)

```
module ex_mixsign .

rem `u8` 은 `i16` 안에 값을 잃지 않고 들어간다.
fn ok_widen input a u8 . input b i16 . output i16 . do return add a. b. . . end .

rem 더 넓은 자리를 결과로 골라도 된다.
fn ok_wider input a u8 . input b i16 . output i32 . do return add a. b. . . end .
```

실행 결과 (output)

```
ok_widen(200, -100) = 100 · ok_wider(200, -100) = 100
```

거부되는 예제 (rejected) — 값을 지키는 넓히기가 없으면 거부된다

소스 코드 (source)

```
module ex_sign .

proc p input a i32 . input b u32 . output i32 . effects none .
do
  return add a. b. . .     rem `u32` 는 `i32` 안에 안 들어간다 – 폭이 같다
end .
```

진단: `E-TYPE-SIGN`

주의 — 넘침이 조용히 감기지 않는다

많은 언어에서 `u8` 의 255 에 1 을 더하면 0 이 된다(감김). 로우엔트는 **트랩**한다. 감김이 필요하면 `wrap_add` 처럼 **그렇게 적힌 연산**을 쓴다 — 감기기를 원했는지 실수였는지가 소스에 남아야 하기 때문이다.

쉬운 말로

"넘치면 멈춘다"가 불편해 보일 수 있다. 그런데 넘친 값으로 계속 계산하면 그 프로그램의 답은 이미 틀렸고, 다만 **어디서 틀렸는지 모르게** 될 뿐이다. 멈추는 쪽이 고칠 수 있는 실패다.

참고 (informative)

트랩이 성능을 해치지 않는 이유는 §6.4.6 에 있다. 계약이 범위를 증명하면 그 검사는 번역 시점에 사라진다 — 곧 **정직하게 적으면 빨라진다**.

6.3.5 비트 연산 (Bitwise operations)

1 비트 연산은 값을 **비트의 나열**로 보고 다룬다. 피연산자와 결과는 정수이며, 부호 있는 정수도 다룰 수 있다.

1′ `bit_and` · `bit_or` · `bit_xor` 의 두 피연산자가 같은 타입이면 결과도 그 타입이고, 한쪽이 리터럴이면 다른 쪽의 타입이다. `bit_not` 과 옮기기 · 돌리기(`shl` · `shr` · `rotl` · `rotr`)의 결과는 (왼쪽) 피연산자의 타입이다. `usize` · `idiff` 와 고정 폭 정수를 한 연산에 대면 번역이 거부된다(`E-TYPE-NOMINAL`, §6.2 (3-2)) — 옮기는 양은 이 규칙에 들지 않는다.

1″ `bit_and` · `bit_or` · `bit_xor` 의 두 피연산자가 폭이나 부호만 다른 정수이면 산술과 같은 규칙을 따른다(§6.3.4 (1)–(2)): 넓히기로 비교할 수 있으면 결과는 넓은 쪽의 타입이고, 그럴 수 없으면 번역이 거부된다(`E-TYPE-SIGN`). `count_ones` · `leading_zeros` · `trailing_zeros` 의 결과는 `a` 의 타입이다.

1a 오른쪽 옮기기 `shr` 는 부호에 따라 뜻이 다르다. 부호 없는 정수에서는 빈자리에 0 이 들어오고(논리 이동), 부호 있는 정수에서는 **부호 비트가 들어온다**(산술 이동). 그래서 음수를 오른쪽으로 옮겨도 음수로 남는다.

2 비트별 논리 연산은 `bit_and` `bit_or` `bit_xor` `bit_not` 이다. 앞의 셋은 두 값을 받고 `bit_not` 은 하나를 받는다.

3 옮기기는 `shl`(왼쪽) 과 `shr`(오른쪽) 이다. 돌리기는 `rotl` 과 `rotr` 이며, 밀려난 비트가 반대쪽으로 돌아 들어온다.

3a 돌리기의 칸 수는 어느 정수 타입이어도 되며, **폭으로 나눈 나머지**만큼 돌린다 — 폭이 8 인 타입을 9 칸 돌리면 1 칸 돌린 것과 같다. 돌리기는 멈추지 아니한다.

4 옮기는 칸 수는 타입의 폭보다 **작아야** 한다. 그렇지 않으면 트랩한다.

4a `shl` · `shr` · `wrap_shl` · `wrap_shr` 의 칸 수는 **부호 없는 정수 타입**이다(폭은 무엇이든 좋다). 부호 있는 정수 타입의 값을 칸 수로 대면 번역이 거부된다(E-SHIFT-SIGNED) — 음수 칸 수에는 하나로 정해진 뜻이 없다. 부호 있는 값을 칸 수로 쓰려면 부호 없는 타입으로 명시적으로 바꾼다. 리터럴은 자리의 타입을 받으므로 그대로 쓴다(음수 리터럴은 (9)).

5 `wrap_shl` 과 `wrap_shr` 은 옮기는 칸 수를 폭 안으로 **감싸서** 쓴다. 폭이 8 인 타입에서 9 칸을 말하면 1 칸으로 읽는다.

> **거부되는 예제 (rejected) — 미는 칸 수는 부호 없는 정수 타입이다**
>
> **소스 코드 (source)**
>
> ```
> module ex_shift_signed .
>
> fn up input a u8 . input n i8 . output u8 . do return shl a. n. . . end .
> ```
>
> 진단: E-SHIFT-SIGNED

6 세는 연산은 `count_ones`(1 인 비트의 수) `leading_zeros`(맨 앞부터 0 이 이어지는 수) `trailing_zeros`(맨 뒤부터 0 이 이어지는 수) 이다.

7 `byte_swap` 은 바이트의 차례를 뒤집는다. 엔디안이 다른 곳과 값을 주고받을 때 쓴다.

> **예제 (example) — 비트별 논리와 뒤집기**
>
> **소스 코드 (source)**
>
> ```
> module ex_bitlogic .
>
> 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 both input a u8 . input b u8 . output u8 . do return bit_or  a. b. . . end .
> fn diff input a u8 . input b u8 . output u8 . do return bit_xor a. b. . . end .
> fn flip input a u8 . output u8 . do return bit_not a. . . end .
> ```
>
> **실행 결과 (output)**
>
> ```
> mask(12,10) = 8 · both(12,10) = 14 · diff(12,10) = 6 · flip(12) = 243
> ```

> **쉬운 말로**
>
> `flip(12)` 이 243 인 것은 `u8` 이라서다 — `0000_1100` 을 뒤집으면 `1111_0011` 이고 그것이 243 이다. 폭이 다른 타입이면 답도 다르다. 비트 연산의 답은 **언제나 타입의 폭에 매여 있다.**

> **예제 (example) — 옮기기와 돌리기**
>
> **소스 코드 (source)**
>
> ```
> module ex_bitshift .
>
> fn up   input a u8 . input n u8 . output u8 . do return shl  a. n. . . end .
> fn down input a u8 . input n u8 . output u8 . do return shr  a. n. . . end .
> fn spin input a u8 . input n u8 . output u8 . do return rotl a. n. . . end .
> fn back input a u8 . input n u8 . output u8 . do return rotr a. n. . . end .
> ```
>
> **실행 결과 (output)**
>
> ```
> up(3,2) = 12 · down(12,2) = 3 · spin(129,1) = 3 · back(3,1) = 129
> ```

쉬운 말로

`spin(129, 1)` 이 3 인 것이 돌리기와 옮기기의 차이를 보여 준다. 129 는 `1000_0001` 이고 왼쪽으로 한 칸 돌리면 맨 앞 1 이 맨 뒤로 돌아와 `0000_0011` = 3 이 된다. 같은 값을 `shl` 로 옮기면 그 1 은 **버려진다.**

주의 — 옮기는 칸 수가 폭 이상이면 트랩한다 — C 와 다른 자리다

폭이 8 인 값을 9 칸 옮기면 처리기가 멈춘다(`E-VM-SHIFT`). 씨(C)에서 같은 일은 **미정의 동작**이고, 프로그램은 계속 돌며 답은 기계가 정한다. 여기서는 계약이다. 감싸는 쪽을 원하면 **이름으로 말한다** — `wrap_shl` · `wrap_shr`. 폭이 8 일 때 9 칸은 1 칸으로 읽힌다(`wrap_shr(12, 9) = 6`). 조용히 감싸는 것과 감싸 달라고 적는 것은 다른 일이다(§1.3).

예제 (example) — 부호가 옮기기의 뜻을 바꾼다

소스 코드 (source)

```
module ex_signed_shift .

rem 부호 있는 정수 – 빈자리에 부호 비트가 들어온다(산술 이동). 칸 수는 부호 없는 타입이다((4a)).
fn s_shr input a i32 . input n u32 . output i32 . do return shr a. n. . . end .

rem 부호 없는 정수 – 빈자리에 0 이 들어온다(논리 이동).
fn u_shr input a u32 . input n u32 . output u32 . do return shr a. n. . . end .

rem 비트별 논리 연산도 부호 있는 정수를 다룬다.
fn s_and input a i32 . input b i32 . output i32 . do return bit_and a. b. . . end .
fn s_not input a i32 . output i32 . do return bit_not a. . . end .
```

실행 결과 (output)

```
s_shr(-8, 1) = -4 · u_shr(4294967288, 1) = 2147483644 · s_and(-8, 12) = 8 · s_not(0) = -1
```

쉬운 말로

-8 과 4294967288 은 32 비트에서 **같은 비트열**이다. 그런데 오른쪽으로 한 칸 옮기면 답이 갈린다 — 앞의 것은 -4, 뒤의 것은 2147483644 다. 비트 연산이라고 해서 타입을 안 보는 것이 아니다. **부호가 뜻을 정한다.**

예제 (example) — 세기와 바이트 뒤집기

소스 코드 (source)

```
module ex_bitcount .

fn ones input a u8 . output u8 . do return count_ones a. . . end .
fn lead input a u8 . output u8 . do return leading_zeros a. . . end .
fn tail input a u8 . output u8 . do return trailing_zeros a. . . end .
fn endian input a u32 . output u32 . do return byte_swap a. . . end .
```

실행 결과 (output)

```
ones(7) = 3 · lead(1) = 7 · tail(8) = 3 · endian(1) = 16777216
```

쉬운 말로

`lead(1)` 이 7 인 것도 폭 때문이다 — `u8` 에서 1 은 `0000_0001` 이므로 앞선 0 이 일곱이다. `endian(1)` 이 16777216 인 것은 `0x00000001` 의 바이트를 뒤집으면 `0x01000000` 이기 때문이다.

8 비트 연산은 **정수만** 받는다 — 비트는 기계 낱말의 성질이므로, 그 성질이 없는 값에는 뜻이 없다. 이 어긋남은 **실행 자리에서** 잡힌다(E-BITOP-TYPE).

> **주의 —**
>
> 번역할 때 잡지 못하는 자리가 여기 있다. 그 사실을 적어 두는 까닭은, 적지 않으면 읽는 사람이 **번역이 지켜 준다고 믿기** 때문이다 — 지키지 않는 것을 지킨다고 믿게 두는 것이 규범이 할 수 있는 가장 나쁜 일이다(§8.6).

9 옮기는 칸수가 그 타입의 폭보다 **작지 않음이 증명되면** 거부된다(E-SHIFT-RANGE).

6.3.6 넘쳤을 때 어떻게 할지 이름으로 고른다

1 `add·sub·mul` 은 넘치면 멈춘다(§6.3.4). 다른 처분이 필요하면 **그 처분이 이름에 적힌 op** 을 쓴다. 곧 처분은 설정이 아니라 **낱말**이다.

표 29 — 처분을 이름으로 고르는 산술

처분	op	넘치면 무엇을 하나
멈춤(기본)	`add` · `sub` · `mul`	덫에 걸린다 — 틀린 값으로 계속 가지 아니한다
감김	`wrap_add` · `wrap_sub` · `wrap_mul`	폭 안에서 돌아 감는다
포화	`sat_add` · `sat_sub` · `sat_mul`	그 폭의 끝값에 멈춰 선다
값으로	`chk_add` · `chk_sub` · `chk_mul`	넘침을 **값으로** 돌려준다(⟦option⟧)
검사 없음	`div_nz`	제수가 0 이 **아님이 타입으로 보장된** 나눗셈. 그 자리에는 검사가 남지 아니한다

1a 넘침 처분을 적은 셈(`wrap_*` · `sat_*` · `chk_*`)은 **정수**의 것이다. 부동소수점 수에 쓰면 번역이 거부된다(E-TYPE-KIND) — 부동소수점 셈에는 넘친 값이 갈 무한대가 있다.

2 나눗셈의 0 도 같다. 제수가 0 이 **아님이 타입으로 보장된** 자리에서는 `div_nz` 를 쓰며, 그 자리에는 검사가 남지 아니한다. 보장된 제수는 셋이다 — 0 이 아닌 정수 리터럴 · (2a) 의 값을 `some_value` 로 꺼낸 것 · 그것으로 묶은 `let`. (2c) 의 `nonzero_or` 가 내는 값과 그것으로 묶은 `let` 도 보장된 제수다. 그 밖의 제수를 `div_nz` 에 주면 번역이 거부된다(E-TYPE-NONZERO). 0 에서 멈추는 나눗셈은 `div` 다.

2a 그 보장을 얻는 길이 `nonzero_of` 다. `nonzero_of <값>` 은 그 값이 0 이 아님을 **값으로** 알린다(⟦option⟧) — 0 이면 없음이다. 한 번 얻은 뒤에는 나누는 자리마다 다시 묻지 아니한다.

> **쉬운 말로**
>
> 「0 으로 나누지 않았음」을 매번 검사하는 대신 **한 번 증명해 값으로 들고 다니는** 것이 이 언어가 계약에서 하는 일과 같다. 검사를 없애는 것은 위험을 무시하는 것이 아니라, 위험이 없음을 **한 자리에서** 확인하고 그 사실을 타입에 실어 나르는 것이다.

2b 넘친 자리올림 · 빌림을 **버리지 않고 값으로** 받으려면 표준 라이브러리 `intops` 를 쓴다(§9) — `add_carry_<폭>` 은 합과 자리올림(0 또는 1)을, `sub_borrow_<폭>` 은 차와 빌림을, `mul_wide_<폭>` 은 곱의 아래 절반 `lo` 와 위 절반 `hi` 를 구조체로 돌려준다(부호 있는 폭은 `mul_wide_s_<폭>` — `lo` 는 부호 없고 `hi` 는 부호 있다). 폭은 부호 없는 `u8` · `u16` · `u32` · `u64` · `usize` 와 부호 있는 `i8` · `i16` · `i32` · `i64` · `idiff` 다. 모두 로우엔트로 적혀 있다(§9.3 (5)).

2c `nonzero_or <값> <대신>` 은 값이 0 이 아니면 그 값을, 0 이면 대신을 낸다. 대신은 값과 같은 타입이고 0 이 **아님이 보장된** 값이어야 한다 (0 이 아닌 정수 리터럴 · `nonzero_of` 의 값을 `some_value` 로 꺼낸 것 · 그것으로 묶은 `let` · 다른 `nonzero_or` 의 결과) — 아니면 번역이 거부된다(E-TYPE-NONZERO). 그러므로 결과도 0 이 아님이 보장되어, `div_nz` 의 제수가 되고 그것으로 묶은 `let` 도 그렇다: `div_nz total. nonzero_or count. 1 . ..` 정수만 받는다(E-TYPE-KIND).

3 폭을 줄이는 일도 처분을 이름으로 고른다.

표 30 — 폭을 바꾸는 op

op	모양	무엇을 하나
`widen`	`widen <타입> <값>`	넓힌다. 값이 보존되므로 처분이 필요 없다
`narrow`	`narrow <타입> <값>`	줄인다. 안 들어가면 **멈춘다**
`narrow_wrap`		안 들어가면 감는다
`narrow_sat`		안 들어가면 끝값에 선다
`narrow_try`		안 들어가면 **값으로** 알린다(⟦option⟧)

> **쉬운 말로**
>
> 왜 처분을 빌드 설정이 아니라 이름으로 두는가. 설정으로 두면 **같은 소스가 빌드마다 다른 답**을 내고, 그 답이 어느 것인지는 소스를 읽어서 알 수 없다. 이름으로 두면 감기기를 **원했는지 실수였는지**가 그 자리에 남는다 — 이 언어가 계약과 효과에서 되풀이하는 것과 같은 원칙이다.

6.3.7 레인을 다루는 op

1 `vec`(§6.2.11)은 산술을 레인마다 따로 하지만, **레인을 가로지르는** 일은 op 으로 따로 있다.

표 31 — 레인을 다루는 op

op	모양	무엇을 하나
`splat`	`splat <값>`	한 값을 모든 레인에 채운다
`load`	`load <조각> <자리>`	메모리에서 레인들을 읽는다
`store`	`store <조각> <자리> <값>`	레인들을 메모리에 쓴다
`load_masked` · `store_masked`	+ 가림막과 기본값	가림막이 켜진 레인만 읽거나 쓴다
`lane_reverse`	`lane_reverse <값>`	레인의 차례를 뒤집는다
`lane_rotate`	`lane_rotate <값> <칸>`	레인을 그만큼 돌린다
`reduce_add` · `reduce_mul`	`reduce_add <값>`	모든 레인을 더한다 · 곱한다
`reduce_min` · `reduce_max`	`reduce_min <값>`	레인 가운데 가장 작은 것 · 가장 큰 것
`lane_avg`	`lane_avg <값> <값>`	레인마다 반올림하는 평균
`native_lanes`	`native_lanes <타입>`	이 기계가 한 번에 다루는 레인 수를 **번역 시점에** 묻는다

> **주의 —**
>
> `lane_avg` 의 값은 **`(a+b+1)>>1` 로 정해져 있다.** 레인이 좁으면 `a+b` 가 그 레인에 넘치므로, 처리기는 더 넓은 자리에서 더한 뒤 좁혀야 한다 — 이 판의 두 뒤끝이 그렇게 한다. 뒷날 이것을 기계 명령 하나로 내리려는 처리기는 **그 명령이 정확히 이 값인지** 확인해야 한다. x86 의 PAVGB/PAVGW 와 ARM 의 VRHADD/URHADD 는 이 값이지만, RISC-V 의 vaaddu 는 반올림 방식이 vxrm 레지스터에 달려 있어 그렇다고 말할 수 없다. 같은 이름이 기계에 따라 다른 값을 내면, 그것은 이 언어가 지키기로 한 것을 놓는 일이다.

1a `splat` 은 레인 수를 **선언된 타입에서** 받는다. 그래서 쓸 수 있는 자리는 벡터 타입을 적은 바인딩의 값 자리뿐이다: `var lim vec 4 u32 splat 5 . ..` 식 안에 바로 적는 것은 적합하지 아니하다(`E-VEC-SPLAT`) — 그 자리에는 레인 수를 말해 주는 것이 없다.

1b `lane_avg` 는 **정수 레인의** 것이다(위의 값이 정수의 셈이다). 부동소수점 레인에 쓰면 번역이 거부된다(`E-TYPE-KIND`, 2026-11-02).

1c 레인을 다루는 op 은 **벡터**를 받는다. 수 하나를 주면 번역이 거부된다(`E-TYPE-KIND`).

1d `splat <리터럴>` 의 리터럴은 레인 하나를 그 레인 타입의 자리로 보고 §6.1.4 (8)(9) 를 따른다 — 부동소수점 레인에 정수 리터럴을 채우면 `E-TYPE-LET`, `f32` 레인에 그 범위를 넘는 부동소수점 리터럴을 채우면 `E-TYPE-WIDTH` 다. 나열 리터럴(`lit vec` · `lit array` · `lit slice`)의 원소도 같다.

2 레인을 가로지르는 op 은 레인 수만큼을 **하나로** 모은다. 결과의 타입은 레인의 타입이다. 부호 있는 레인은 부호로 더하고 곱하며, `reduce_add` · `reduce_mul` 의 값이 그 타입에 들어가지 않으면 멈춘다 (§6.3.4 (3)).

> **참고 (informative)**
>
> 이 표의 앞자리에는 `sum` 과 `sum_fast` 가 «모든 레인을 더한다» 로 적혀 있었다. 그 이름으로 그 일을 하는 op 은 **없었다** — 레인을 더하는 것은 `reduce_add` 이고, `sum` 이라는 이름은 처리기에서 **조각의 부동소수점 합** (§6.3.7.1)에 쓰이고 있었다. 한 이름이 두 가지를 뜻하고 있었으므로 이름을 갈랐다(2026-09-17).

6.3.7.1 조각을 더한다

1 조각(⟦slice⟧)에 담긴 **부동소수점 수**를 모두 더하는 빌트인 연산은 둘이며, 이름이 **어떻게 더하는지**를 말한다. 표준 라이브러리 `math` 가 같은 이름의 op 으로 감싼다(`math.sum_neumaier` · `math.sum_seq`).

1a `sum_neumaier <조각>` 은 더하며 **잃은 자리를 따로 모아 되돌린다.** 그래서 오차가 항의 개수에 매이지 아니한다.

1b `sum_seq <조각>` 은 앞에서 뒤로 한 번 더한다. 빠르고, 오차가 항의 개수에 **비례한다.**

> **거부되는 예제 (rejected) — 부동소수점 합을 정수 자리에 돌려준다**
>
> **소스 코드 (source)**
>
> ```
> module ex_slice_sum .
>
> use math .
>
> fn total input b slice u8 . output u64 .
> do
>   var xs slice f64 view_array f64 b. . .
>   return math.sum_neumaier xs. . .      rem 결과는 부동소수점 수인데 머리는 `u64` 라고 적었다
> end .
> ```
>
> 진단: `E-TYPE-RETURN`

2 두 op 의 결과는 **부동소수점 수**다. 정수로 적은 자리에 그대로 돌려주면 거부된다 (`E-TYPE-RETURN`).

> **쉬운 말로**
>
> 왜 이름이 방법을 말하는가. `sum_seq` 로 `[1e16, 1.0, -1e16]` 을 앞에서 뒤로 더하면 0 이 나오고, `sum_neumaier` 로 더하면 1 이 나온다 — 둘 다 틀린 답이 아니라 **다른 약속**이다. 이름이 `sum` 하나뿐이면 어느 약속을 받았는지 부르는 쪽이 알 수 없고, 빠른 쪽으로 바꾸는 일이 **답을 조용히 바꾸는 일**이 된다. 비용과 오차는 이 언어에서 이름이 지는 몫이다.

> **주의 —**
>
> `native_lanes` 는 **답을 바꾸는 값이 아니다.** 레인 수는 타입의 일부이므로(§6.2.11) 이 물음으로 얻은 수를 쓰든 손으로 적은 수를 쓰든 답은 같다. 다만 기계에 맞는 수를 고르면 **한 번에 더 많이** 셈한다.

6.3.8 크기를 묻는다 — `size_of`

1 `size_of <타입>` 은 그 타입 값 하나가 차지하는 바이트 수를 **번역 시점에** 준다. 그 수의 타입은 `usize` 다 — `align_of` 가 주는 정렬, `native_lanes` 가 주는 레인의 수도 같다.

2 크기를 알 수 없는 타입에는 쓸 수 없다.

> **쉬운 말로**
>
> 이 물음이 없으면, 타입을 매개변수로 받는 코드는 **가장 넓은 폭을 가정할** 수밖에 없다. 그러면 한 바이트 짜리를 담는 그릇도 여덟 배를 쓴다. 크기를 물을 수 있어야 매개변수를 받는 코드가 비용을 정직하게 낼 수 있다.

6.3.9 초월 함수

1 빌트인 연산 `sin` · `cos` · `exp` · `log` · `pow` · `sqrt` · `floor` · `ceil` · `round` · `fmod` 는 **부동소수점 전용**이다. 표준 라이브러리 `math` 가 같은 이름의 op 으로 감싼다(`math.sqrt x.`). 응용 프로그램은 그 op 을 부른다.

1a `abs` 는 부동소수점 수와 **부호 있는 정수**를 받는다. 결과의 타입은 피연산자의 타입이다. 부호 있는 정수의 가장 작은 값에서는 멈춘다 — 절댓값이 그 폭에 없다(§6.3.4 (3)). 부호 없는 정수에 쓰면 번역이 거부된다(`E-TYPE-SIGN`) — 그 값이 이미 제 절댓값이다.

> **거부되는 예제 (rejected) — 부호 없는 정수의 `abs`**
>
> **소스 코드 (source)**
>
> ```
> module ex_abs_unsigned .
>
> fn f input a u32 . output u32 . do
>   return abs a. . .     rem 부호 없는 값은 이미 제 절댓값이다
> end .
> ```
>
> 진단: `E-TYPE-SIGN`

2 이들은 권한을 요구하지 아니하며, **운영체제가 있는 기계에서만** 쓸 수 있다.

> **모호 (open issue) — M-0007 — 초월 함수 결과의 마지막 비트**
>
> `sin` · `cos` · `exp` · `log` · `pow` 의 결과가 마지막 비트까지 어떤 값인지 이 문서는 정하지 아니한다. 그런데 그것이 처리기의 재량 가운데 어느 갈래인지(§4.3) 표시되어 있지 아니하다. 풀이 ㄱ: 자유 선택으로 둔다 — 고를 수 있는 범위는 «참값에 가장 가까운 두 부동소수점 가운데 하나» 처럼 적는다. 풀이 ㄴ: 구현 정의로 둔다 — 처리기가 쓰는 셈 라이브러리를 문서로 적는다. 풀이 ㄷ: 정확히 반올림한 값으로 정한다. 처리기는 지금 기계의 셈 라이브러리가 내는 값을 그대로 낸다.

> **주의 —**
>
> 이 함수들의 결과가 마지막 비트까지 어떤 값인지는 이 문서가 정하지 아니한다. 기계와 그 아래 셈 라이브러리가 정한다. 마지막 비트가 걸린 계산이라면 **그 사실을 알고** 써야 한다.

6.3.10 참거짓은 수가 아니다

1 `and` · `or` · `not` 은 **참거짓만** 받는다. 정수를 건네면 거부된다 (`E-TYPE-LOGICAL`).

2 조건이 오는 자리(§6.5)도 마찬가지로 참거짓만 받는다(`E-TYPE-COND`).

3 곧 `0` 이 거짓이고 그 밖이 참이라는 약속은 **없다.** 무엇을 묻는지는 그 자리에 적어야 한다 — `gt a. 0 .` 처럼.

4 참거짓은 셈(`add` · `sub` · `mul` · `div` · `mod` · `min` · `max` · `abs` 와 넘침 처분을 적은 셈) · 크기 견줌(`lt` · `le` · `gt` · `ge`) · 비트 연산(`bit_and` · `bit_or` · `bit_xor`)의 피연산자가 될 수 없다(`E-TYPE-KIND`). 두 참거짓은 `eq` · `ne` 로 견준다. 참거짓을 수로 세려면 `cast u8 b. .` 로 적는다(§6.2.16 (4)).

쉬운 말로

정수를 조건 자리에 그냥 두는 언어에서는 `if x` 가 *"x 가 0 이 아니다"* 인지 *"x 가 있다"* 인지 *"x 가 참이다"* 인지 읽는 사람이 **문맥으로 짐작**한다. 짐작이 필요한 자리는 짐작이 틀리는 자리이기도 하다. 묻는 것을 적게 하면 그 자리가 사라진다.

거부되는 예제 (rejected) — 정수는 조건이 아니다

소스 코드 (source)

```
module ex_cond .

fn f input a u8 . output u8 .
do
  if a. do return 1 . end .     rem 무엇을 묻는지 적는다 — `gt a. 0 .`
  return 0 .
end .
```

진단: E-TYPE-COND

6.3.11 폭과 레인은 타입의 일부다

1 레인 수가 다른 두 vec 은 서로 다른 타입이며 섞이지 아니한다(E-TYPE-LANES).

2 가림막을 받는 자리에는 가림막만 올 수 있다(E-TYPE-MASK).

3 묶음의 칸 배치가 같아도 **이름이 다르면 다른 타입**이다(E-TYPE-WRAP) — 같은 배치가 같은 타입을 만들지 아니한다(§8.9).

4 좁힌 범위(§6.2.14)를 벗어난 값을 그 자리에 두면 거부된다(E-TYPE-RANGE). **어떤 값도 만족시킬 수 없는** 범위를 적는 것도 마찬가지다 — 그것은 조건이 아니라 거짓말이다.

거부되는 예제 (rejected) — 레인 수가 다른 둘은 다른 타입이다

소스 코드 (source)

```
module ex_lanes .

fn f input a vec 4 u8 . input b vec 8 u8 . output vec 4 u8 .
do
  return add a. b. . .
end .
```

진단: E-TYPE-LANES

6.3.12 `expr` 섬 안의 규칙

1 `expr` 섬(§6.3.2) 안의 토막은 저마다 폼 하나다 — 이름이면 `a.`, 호출이면 제 점으로 닫힌 폼이다 (`expr twice a. . + 1 .`). 폼이 제 끝을 스스로 가지므로 호출을 괄호로 묶을 필요가 없다. 섬은 제 점으로 닫힌다.

2 섬 안에는 **중위 연산자만** 있다. 앞에 붙는 한 자리 연산자는 없다 (E-EXPR-UNARY) — 부호를 뒤집으려면 그렇게 적힌 op 을 부른다(`neg a. .`).

거부되는 예제 (rejected) — 섬 안에는 앞에 붙는 연산자가 없다

소스 코드 (source)

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

> 진단: `E-EXPR-UNARY`

> **쉬운 말로**
>
> 섬을 둔 까닭은 산술을 **눈에 익은 차례로** 읽게 하려는 것이지, 두 번째 문법을 들이려는 것이 아니다. 그래서 섬 안에서 할 수 있는 일은 좁고, 좁은 만큼 그 안의 뜻은 흔들리지 아니한다.

6.4 선언과 계약 (Declarations and contracts)

6.4.1 op 의 선언

1 op (op) 은 이름과 계약을 가진 실행 단위다. op 은 두 갈래로 갈린다.

> **fn** (pure op)
>
> 아무 효과도 내지 않는 op. 같은 입력에 언제나 같은 결과를 낸다. 바깥세상을 건드리지 않으므로 순서를 바꾸어 실행해도 결과가 같다.

> **proc** (effectful op)
>
> 효과를 내는 op. 무슨 효과를 내는지 자기 계약에 적어야 하며(§7.1), 적은 것보다 많은 효과를 내면 번역이 거부된다.

2 갈래는 **언제나 적는다.** 기본값이 없다 — `fn` 인지 `proc` 인지는 이름 앞에 적혀 있다.

3 op 의 선언은 이름, 입력, 출력, 효과, 계약, 본문의 차례로 이루어진다. 입력은 `input`, 출력은 `output` 으로 적는다. 차례는 **앞의 것이 뒤의 것에 쓰이도록** 놓였다:

- 이름 붙은 약속(`satisfies`)은 이름 바로 뒤에 온다 — 이 op 이 **무엇인지**를 먼저 말한다.
- 번역 시점 입력(`comptime`)이 다음이다 — 뒤따르는 입력·출력의 타입이 그 이름을 쓴다.
- 권한 입력이 데이터 입력보다 먼저다 — 누가 허락했는지가 무엇을 받는지보다 먼저 보인다.
- 출력은 입력 뒤다 — 출력 타입은 입력의 타입 매개변수를 쓸 수 있다.
- 효과는 출력 뒤, 계약 앞이다 — 앞에 받은 권한으로 **무엇을 하는지**를 적는다.
- 계약은 입력 조건(`requires`) · 출력 약속(`ensures`) · 실패(`errors`) · 시험(`tests`) 차례다.

3a 머리의 절은 **한 가지 차례로만** 적는다. 앞에서 뒤로: `satisfies`·`lowdoc` · `vector`·`priority` · `comptime` 입력 · 권한·영역 입력(타입이 `cap …`·`region …`) · `use` · 데이터 입력 · `output` · `effects` · `link`·`variadic` · `asm`·`absorbs`·`reference`·`why` · `access`·`inplace`·`invalidates`·`parallel`·`reduce` · `requires` · `ensures` · `errors` · `tests` · `schedule`. 같은 자리의 절끼리는 적힌 차례를 지킨다. trait 의 메서드 서명(§6.11.2)과 actor 안의 `proc` 도 같은 차례를 따른다. 이 차례를 어기면 번역이 거부된다(`E-CLAUSE-ORDER`). 입력의 차례는 부르는 쪽 인자의 차례이기도 하다 — 그러므로 권한은 언제나 데이터보다 **먼저** 건넨다.

3b 서식기(`--fmt`)는 입력이 아닌 절과 `use` 절을 이 차례로 옮겨 적는다. 입력끼리의 차례는 옮기지 아니한다 — 입력을 옮기면 호출의 뜻이 바뀌기 때문이다.

3c 머리의 절은 **저마다 자기 점으로 닫는다** — 문장과 같다(§6.1.6 (2)). 절 낱말은 앞 절을 닫지 아니하고, 개행도 닫지 아니한다. 점 없이 다음 절 낱말이나 `do` 를 만나면 번역이 거부된다(`E-DOT-MISSING`). 절의 목록이 곧 머리이므로, 이름 뒤에 오는 첫 낱말부터 절이다 — 이름이 절 낱말과 같은 철자여도 (`proc link …`) 이름으로 읽는다.

3d (3a) 의 차례에 나온 절 낱말의 뜻과 정의 자리는 다음과 같다.

- `satisfies` — 이 op 이 지키는 이름 붙은 약속(§6.4.5 · §6.11.2). `lowdoc` — 사람이 읽는 설명(§6.4.3).
- `vector` · `priority` — 이 op 을 인터럽트 처리기로 만든다: 받는 인터럽트의 번호와 그 급함(§7.7).
- `comptime` 입력 — 번역 시점에 정해지는 매개변수(§6.8). 권한 입력 — 효과를 낼 자격(§7.2).
- `use` — 이 op 이 쓰는 얼로케이터(§8.14).
- `link` · `variadic` — C 쪽의 이름 · 개수가 정해지지 않은 인자(§7.4).
- `asm` — 기계 명령을 직접 적는 자리(§6.9.1).

- `absorbs` · `reference` · `why` — `unsafe` 가 번지는 것을 여기서 멈춘다는 선언과, 그 근거가 되는 순수한 판 · 가정한 것(§7.2.1).
- `access` · `inplace` · `invalidates` — 무엇을 어떤 모드로 건드리는가 · 제자리 쓰기 · 무효로 하는 것(§6.4.3 · §8.12).
- `parallel` · `reduce` — 나누어 도는 되풀이와, 걸음들이 모으는 자리(§6.5.10).
- `requires` · `ensures` · `errors` · `tests` — 계약의 절(§6.4.3).
- `schedule` — 시험(`test`) 머리에만 온다. `schedule explore_interleavings .` 은 메시지가 배달되는 **차례를 모두** 돌려 보고 결과가 차례마다 같은지 본다. `limit <수>` 를 붙이면 그 수까지만 돈다. 차례에 따라 결과가 갈리면 그 시험은 실패한다(`E-SCHED-NONDET`). 뜻은 §6.4.13 (4) 가 정한다.

3e **출력에 이름을 붙일 수 있다** — `output <이름> <타입> ..` 그 이름은 몸의 지역이고 **0 으로 시작한다**: 수는 `0` · `0.0`, `bool` 은 `false`, `option` 은 `none`, 구조체는 칸마다 그 칸의 0 이다. 몸은 그 이름에 쓰고 `return <이름> .` 으로 돌려준다(값을 돌려주는 다른 op 과 같이 모든 길이 `return` 으로 끝나야 한다 — §6.5.5 (3)).

- 가름: 이름 공간이 하나이므로(§6.10) `output` 뒤의 첫 낱말이 **타입 이름이면** 출력 전체가 타입이고, 타입 이름이 아니고 그 뒤가 타입 하나이면 그 낱말이 결과의 이름이다.
- 0 이 정해지지 않은 타입(슬라이스 · `result` · `owned` · enum · newtype · 제네릭 매개변수, 그런 칸을 가진 구조체)에는 이름을 붙일 수 없다(`E-RESULT-NOZERO`). 그런 출력은 이름 없이 적고 값을 돌려준다.
- 몸이 없는 op(`extern` · trait 의 메서드 서명)은 결과에 이름을 붙일 수 없다(`E-RESULT-NAMED`).
- 배열을 돌려주는 출력(`output [<이름>] array <길이> <타입> .`)은 **부르는 쪽이 자리를 준다** — §6.2.6 (1c).
- `ensures` 는 결과를 여전히 `ret` 으로 가리킨다(§6.4.12). 결과의 이름은 몸 안에서만 보인다.

> **예제 (example) — 이름 붙은 결과는 0 에서 시작한다**

소스 코드 (source)

```
module ex_named_result .

struct span do
  lo u64 .
  hi u64 .
  seen bool .
end .

export fn bounds input xs slice u64 . output s span .
do
  for x xs. do
    if or not field s. seen . . lt x. field s. lo . . . do set field s. lo . x. . end .
    if gt x. field s. hi . . do set field s. hi . x. . end .
    set field s. seen . true .
  end .
  return s. .
end .
```

> **거부되는 예제 (rejected) — 0 이 정해지지 않은 타입에 이름을 붙였다**

소스 코드 (source)

```
module ex_named_nozero .

fn first input xs slice u8 . output r slice u8 .
do
  return xs. .
end .
```

진단: `E-RESULT-NOZERO`

예제 (example) — op 의 선언

소스 코드 (source)

```
module ex_op .

export fn twice input n u32 . output u32 .
  requires le n. 2147483647 . .
do
  return mul n. 2 . .
end .
```

거부되는 예제 (rejected) — 출력을 입력보다 앞에 적는다

소스 코드 (source)

```
module ex_clause_order .

fn twice output u32 .
  input n u32 .
  requires le n. 2147483647 . .
do
  return mul n. 2 . .
end .
```

진단: `E-CLAUSE-ORDER`

거부되는 예제 (rejected) — 머리의 절에 점이 없다

소스 코드 (source)

```
module ex_clause_dot .

fn twice input n u32 output u32 .
do
  return mul n. 2 . .
end .
```

진단: `E-DOT-MISSING`

쉬운 말로

`export` 는 다른 모듈이 이 op 을 쓸 수 있게 한다는 표시다. 붙이지 않으면 그 모듈 안에서만 쓴다.

6.4.2 계약이란 무엇인가

1 계약 (contract) 은 op 이 자기 입력과 출력에 대해 **스스로 적는 약속**이다. 계약은 주석이 아니라 **검사되는 문장**이다.

2 계약은 세 곳에서 쓰인다.

3 가) **처리기가 사실로 쓴다** — 계약이 참임을 전제로 검사를 없앤다(§6.4.5).

4 나) **실행 중에 강제된다** — 처리기가 증명하지 못한 계약은 실행 중에 확인되며, 깨지면 트랩한다. 다만 남은 검사를 실제로 둘지는 **빌드 모드**가 정한다(§6.4.8).

5 다) **읽는 사람에게 말한다** — 이 op 을 부르려면 무엇을 지켜야 하는지가 본문을 열지 않고도 보인다.

6 계약이 깨지는 자리는 둘이고, 진단이 **누구의 잘못인지** 가른다.

표 32 — 계약이 깨졌을 때

절	언제 확인되나	깨지면 누구의 잘못인가
`requires`	들어올 때	**부르는 쪽** — 지켜야 할 조건을 안 지켰다
`ensures`	나갈 때	**이 op** — 자기 약속을 어겼다(부르는 쪽은 거짓말을 들었다)

> **쉬운 말로**
>
> 이 구별이 실제로 값을 한다. 프로그램이 멈췄을 때 "내가 잘못 불렀나, 저 함수가 잘못 만들어졌나"를 진단이 바로 말해 준다.

> **주의 — 계약은 비용이 아니라 자산이다**
>
> "검사를 적으면 느려진다"고 생각하기 쉽다. 로우엔트에서는 반대다. 계약을 적으면 처리기가 그것을 **사실로 써서** 실행 중 검사를 없앤다. 적지 않으면 처리기가 모르므로 검사가 남는다 — 곧 **정직하게 적는 쪽이 빠르다**.

6.4.3 계약의 여섯 절

1 계약은 다음 여섯 절로 이루어진다. 모든 절이 필수는 아니다.

> **requires** (precondition)
>
> op 에 들어오는 순간 참이어야 하는 조건. 부르는 쪽이 지킬 책임이 있다. 들어올 때 확인되며, 깨져 있으면 트랩한다.

> **ensures** (postcondition)
>
> op 이 정상으로 끝날 때 참인 조건. 만드는 쪽이 지킬 책임이 있다. 돌려주는 값을 `ret` 이라는 이름으로 가리킬 수 있다.

> **errors** (error condition)
>
> op 이 실패하는 경우와 그때의 오류 값. 어떤 조건에서 어떤 오류가 나오는지를 적는다. 조건은 op 에 **들어올 때의 값**으로 읽는다 — ⟦(1c)⟧.

> **effects** (effect declaration)
>
> 이 op 이 내는 효과의 목록(§7.1). 아무 효과도 내지 않으면 `effects none` 이다.

> **access** (access declaration)
>
> 이 op 이 무엇을 어떻게 건드리는지 — `access <이름> <모드>` . 로 적는다. 모드는 아래 표의 일곱 가지가 전부이며, 그 밖의 낱말은 거부된다(E-CONTRACT-MODE).

표 33 — access 의 모드 — 닫힌 집합 일곱

모드	지금 강제되나	뜻
`shared_read`	**예**	이 자리에 **쓰지 않는다**. 쓰면 거부된다

`write_only`	**예**	이 자리를 **읽지 않는다**. 읽으면 거부된다
`sequential`	아니오	차례대로 훑는다
`random`	아니오	아무 자리나 짚는다
`streaming`	아니오	한 번 지나가고 다시 안 본다
`tiled`	아니오	덩이로 나눠 훑는다
`read_mostly`	아니오	거의 읽고 드물게 쓴다

> **tests** (examples)
>
> 이 op 의 쓰임새를 보이는 예. 처리기는 이것을 시험으로 돌릴 수 있다.

1a `tests <이름>…` 절은 이 op 을 시험하는 **op 의 이름**을 하나 이상 적는다. 적은 이름은 그 자리에서 볼 수 있는 op 이어야 하며, 없으면 거부된다(`E-CONTRACT-UNDEF`). 시험 블록(§6.4.13)의 이름은 op 이 아니므로 여기 적을 수 없다. `tests` 절은 시험을 **정의하지도 돌리지도 아니한다** — 이미 있는 것을 **가리킬 뿐**이다. 가리킨 op 이 없어지면 머리가 그것을 알린다.

1b `lowdoc` 절은 이 op 의 설명을 적는다. 한 줄이면 `lowdoc "<글>"` . 이고, 여러 줄이면 `lowdoc text <끝 표시>` 로 열어 그 표시가 홀로 있는 줄까지가 글이다. 이 글은 문서 산출물 (`--doc`)이 읽으며, 번역되는 프로그램의 뜻에는 영향을 주지 아니한다.

1c `errors` 절의 조건은 op 에 **들어올 때의 값**으로 읽는다.

1c1 몸통이 그 오류를 **한 번도 내지 아니하면** 처리기는 그것을 알린다 (`W-ERRORS-UNRAISED`). 절은 «이 op 이 이렇게 실패할 수 있다» 는 약속으로 읽히고 부르는 쪽이 그것을 보고 갈래를 짓는데, 갈 수 없는 갈래는 **읽는 사람이 살아 있는 갈래와 가릴 수 없는 죽은 코드**가 된다. 남의 `result` 를 그대로 넘기는 op 은 그 오류를 내는 것으로 본다(`try · return <부름>`).

1d 그러므로 그 조건에 설 수 있는 이름은 **들어올 때와 나갈 때가 같은 것**뿐이다 — 곧 바뀌지 않는 입력과 모듈 상수다. 몸통이 바꿀 수 있는 이름(액터의 상태 칸 · 모듈 `var` · `mut` 자리의 인자)을 조건에 적으면 거부된다(`E-ERRORS-STATE`).

> **쉬운 말로**
>
> 왜 들어올 때인가. `errors` 는 ⟦(6)⟧ 의 표에서 `requires` 와 같은 쪽에 선다 — **부르는 쪽이 무엇을 잘못했는가**를 말하는 절이다. 부르는 쪽이 한 일은 **건넨 것**뿐이므로, 그 잘못을 잴 수 있는 값도 건넨 것뿐이다. 나갈 때 값으로 읽으면 성공한 실행이 스스로를 고발한다 — 잔액 50 에서 30 을 빼고 성공한 op 이, 나갈 때 다시 읽힌 「잔액보다 많다」 조건 때문에 "내야 할 오류를 안 냈다"가 된다.

> **주의 —**
>
> (1d) 는 넉넉히 거절한다. `mut` 자리의 슬라이스는 길이가 바뀌지 아니하므로 `len` 만 읽는 조건은 사실 들어올 때와 나갈 때가 같지만, 처리기는 조건이 그 값의 **어느 부분**을 읽는지 가리지 아니한다. 가릴 수 없는 것을 가리는 척하기보다 선을 굵게 긋고 그 사실을 적는 편이 낫다 — 몸통 안에서 `guard` 로 적으면 된다.

2 계약에 쓰는 조건은 **순수한 식**이어야 한다 — 효과를 내는 것을 계약에 적을 수 없다.

3 `shared_read` 와 `write_only` 는 **읽기·쓰기의 규율**이라 처리기가 지금 강제한다. 곧 `shared_read` 라 적고 그 자리에 쓰면 거부되고, `write_only` 라 적고 읽으면 거부된다 (`E-ACCESS-MODE`). 나머지 다섯은 쓰임새를 알리는 **힌트**이며, 처리기는 그것을 아직 쓰지 않는다는 것을 **말한다**(`W-NOT-YET`). 조용히 무시하지 아니한다.

4 강제되지 않는다고 해서 적어도 좋다는 뜻은 아니다. 모드는 이 op 이 무엇을 하는지에 대한 **주장**이며, 처리기가 언젠가 그 주장을 물을 수 있다.

6.4.4 계약을 쓰는 법

조건은 전위 표기로 적는다. 비교와 논리 연산, 산술, `len`, `idx` 를 쓸 수 있다.

`requires` 가 여럿이면 **모두** 참이어야 한다.

슬라이스의 **모든 원소**에 대한 조건은 다음 넷으로 적는다. `elem_lt <슬라이스> <값>` 은 "모든 원소가 그 값보다 작다" 는 뜻이며, `elem_le`·`elem_gt`·`elem_ge` 도 같다.

이 넷이 있는 까닭은 계약이 **하나씩 도는 것을 적을 수 없기** 때문이다. 조건은 순수한 식이고 되풀이는 문장이므로, 원소 전체를 말하려면 그것을 말하는 낱말이 있어야 한다.

> **예제 (example) — 계약**

소스 코드 (source)

```
module ex_contract .

rem 이 op 은 슬라이스에서 두 바이트를 읽어 큰 수 하나를 만든다.
export fn read_pair input data slice u8 . output u32 .
  requires ge len data. . 2 . .
  ensures le ret. 65535 . .
do
  let hi u32 widen u32 idx data. 0 . . .
  let lo u32 widen u32 idx data. 1 . . .
  return add mul hi. 256 . lo. . .
end .
```

> **쉬운 말로**

위 예제를 읽는 법: "이 함수는 길이가 *2* 이상인 바이트 줄을 받아야 하고*(requires),* 돌려주는 값은 *65535* 이하임을 약속하며*(ensures),* 바깥세상은 아무것도 건드리지 않는다*(effects none)."* — 본문을 열지 않고도 이만큼을 알 수 있다.

6.4.5 계약에 이름 주기

여러 op 이 같은 조건을 요구하면, 그 조건에 이름을 줄 수 있다. `contract` 로 적는다.

op 은 `satisfies` 로 그 이름을 적어 그 계약을 갖춘다.

> **예제 (example) — 이름 붙인 계약**

소스 코드 (source)

```
module ex_named .

rem 여러 op 이 같은 조건을 요구하면 그 조건에 이름을 줄 수 있다.
contract nonneg do
  requires ge a. 1 . .
end .

fn half satisfies nonneg . input a u8 . output u8 .
do
  return div a. 2 . .
end .
```

> **쉬운 말로**

같은 조건을 여러 곳에 손으로 되풀이하면, 하나를 고치고 다른 하나를 잊는 일이 생긴다. 이름을 주면 고칠 자리가 하나가 된다.

6.4.6 계약이 검사를 없앤다

1 처리기는 계약을 **사실로 삼아** 값이 가질 수 있는 범위를 좁힌다. 그 범위가 안전을 보이면 그 검사를 없앤다.

2 없앨 수 있는 검사는 다음이다 — 넘침, 0 으로 나누기, 좁히기, 슬라이스 경계.

3 계약이 **강제되지 않으면 사실로 쓰지 아니한다.** 곧 확인 없이 믿는 일은 없다. 확인 없이 믿고 검사를 없애면 그것은 빨라진 것이 아니라 틀린 것이다.

4 처리기는 없애지 못한 검사가 어디에 남았는지 알릴 수 있다. 알리지 않는 성능 모형은 그 자체로 소스 밖의 지식이 된다(§1.3).

> **예제 (example) — 계약이 검사를 없앤다**
>
> **소스 코드 (source)**
>
> ```
> module ex_elim .
>
> rem 계약이 없으면 넘침 검사가 남는다.
> export fn bare input a u8 . output u8 .
> do
>   return add a. 1 . .
> end .
>
> rem 계약이 있으면 처리기가 증명하고 검사를 없앤다.
> export fn proven input a u8 . output u8 .
>   requires le a. 200 . .
> do
>   return add a. 1 . .
> end .
> ```

> **참고 (informative)**
>
> 두 op 은 본문이 같다. 다른 것은 계약 한 줄뿐이며, 그 한 줄이 실행 중 검사 하나를 없앤다. 이것이 이 언어에서 계약이 하는 가장 큰 일이다.

> **도해 (diagram) — 계약이 진입에서 사실이 되어 넘침 검사를 지운다**
>
> ```
> proven 250  ⟶ requires le a 200   entry check fails → E-VM-CONTRACT (caller's fault)
> proven 150  ⟶ requires le a 200   passes → from here on a ∈ [0, 200] is a fact
>               add a 1             a + 1 ∈ [1, 201] ⊂ u8 [0, 255] → overflow check removed
> bare 255    ⟶ add a 1             no fact about a → the check stays → E-VM-OVERFLOW
> ```

6.4.7 빌드 모드가 남은 검사를 정한다

1 빌드 모드 (build mode) 는 소스에 `build <모드> .` 로 적는다. 증명된 계약은 어느 모드에서도 검사가 남지 아니한다. 정하는 것은 **증명하지 못한 계약**의 처분이다.

1a 모드는 **닫힌 넷**이다 — `debug` · `test` · `release_safe` · `release_fast`. 그 밖의 낱말은 거부된다(E-BUILD-MODE).

2 `debug` 는 남은 검사를 두고, 깨지면 무엇이 깨졌는지 말하며 멈춘다 — 깨진 검사의 종류(`requires` · `ensures` · `errors` 절)와 그 검사를 가진 op 의 이름, 그리고 그 절의 소스 줄을 알린다. `test` 도 그렇게 알린다.

2a `test` 는 `debug` 와 같이 검사를 둔다. 시험(§6.4.13)을 돌리는 것은 이 모드가 아니라 **시험을 돌리라는 요청**이다 — 그 요청은 어느 모드에서든 시험을 돌리고, `test` 모드라도 보통의 실행은 시험을 돌리지 아니한다. 시험 안의 `expect` 가 거짓이면 그 시험은 실패한다(E-TEST-FAIL) — 처리기의 잘못이 아니라 **적은 사람의 주장이 틀린 것**이다.

3 `release_safe` 는 남은 검사를 두고, 깨지면 멈춘다. 멈출 때 알리는 것은 깨진 검사의 종류를 적은 **짧은 고정 문구**다 — 계약의 글(소스의 식)은 실행 파일에 싣지 아니한다. `debug` 와 다른 점은 (4) 대로 `debug` 등급의 계약을 없애는 것이다.

4 `release_fast` 는 증명하지 못한 `requires` 의 검사를 **없앤다** — op 에 적은 것도, `satisfies` 로 들인 명명 계약의 것도. `release_safe` 는 `debug` 등급의 `requires` 검사를 없앤다. 어느 모드도 `ensures` 의 출구 검사는 없애지 아니한다.

4a 모드가 검사를 없앤 계약은 **사실이 아니다.** 처리기는 그 계약을 근거로 다른 검사를 지우지 아니한다 — 그 op 은 그 계약이 **적혀 있지 않았던 것처럼** 돈다. 그러므로 계약을 어긴 부름이 그 op 안에서 넘치거나 범위를 벗어나면, 그 셈의 검사가 멈춘다. `assume` 이 사실이 아닌 것(§6.4.9)과 같은 까닭이다.

4b 처리기가 셈마다 넣는 검사 — 넘침 · 0 으로 나누기 · `MIN / -1` · 첨자 · 미는 칸 수 · 좁히기 · 태그 — 는 어느 모드에서도 없어지지 아니한다. 그 검사는 증명될 때에만 없어진다.

5 곧 `release_fast` 는 **신뢰 경계**(§4.4)를 하나 더 여는 것과 같다 — 계약이 참임을 사람이 약속하는 것이다. 약속이 틀리면 프로그램은 계약 없이 쓴 프로그램과 같게 돈다. 정의되지 않은 동작으로 가지는 아니한다.

> **거부되는 예제 (rejected) — 빌드 모드는 닫힌 넷이다**

소스 코드 (source)

```
module ex_build_mode .

build nosuchmode .

fn f output u8 . do return 1 . end .
```

진단: `E-BUILD-MODE`

6.4.8 이름과 범위

1 이름은 네 겹으로 중첩된다 — 모듈, op, 블록, 그 안의 블록.

2 한 이름은 한 범위에서 정확히 한 대상을 가리킨다. 이미 살아 있는 이름을 다시 선언하여 **가릴 수 없다.** 그렇게 하면 번역이 거부된다.

3 가릴 수 없는 것은 다음 넷이다 — 모듈에 있는 이름, 기본 연산의 이름(§6.1.3), 같은 op 의 매개변수 이름, 그리고 바깥 블록에서 아직 살아 있는 지역 이름.

4 같은 블록에서 같은 이름을 두 번 선언하는 것도 적합하지 아니하다. 두 번째 선언은 첫 번째를 **대신하지 않는다** — 같은 글자가 두 대상을 가리키게 될 뿐이다.

5 블록을 벗어나면 그 블록에서 선언한 이름은 사라진다. 따라서 나란한 두 블록이 같은 이름을 각각 선언하는 것은 적합하다 — 두 이름이 동시에 살아 있지 않기 때문이다.

5a 값 자리에 적은 이름은 그 자리에서 살아 있는 대상 — 지역 · 입력 · op · 상수 · 열거의 갈래 — 을 가리켜야 한다. 아무것도 가리키지 않으면 번역이 거부된다(`E-IR-UNDEF`). 처리기는 그 이름을 짚어 말한다.

5b 지역은 **선언한 뒤에야** 산다 — 선언보다 앞에서 쓰면 (5a) 대로 거부된다. 모듈에 선언한 이름(op · 상수 · 타입)은 선언의 차례와 상관없이 그 모듈 어디에서나 산다: 뒤에 선언한 op 을 앞에서 불러도 된다.

> **거부되는 예제 (rejected) — 아무것도 가리키지 않는 이름**

소스 코드 (source)

```
module ex_undeclared .

fn f input a u64 . output u64 . do
```

```
  return add a. b. . .
end .
```

진단: `E-IR-UNDEF`

거부되는 예제 (rejected) — 모듈 이름을 가릴 수 없다

소스 코드 (source)

```
module ex_shadow .

let g u32 7 .

proc p output u32 . effects none .
do
  let g u32 1 .     rem 모듈의 `g` 를 가린다
  return g. .
end .
```

진단: `E-NAME-SHADOW`

거부되는 예제 (rejected) — 매개변수를 가릴 수 없다

소스 코드 (source)

```
module ex_shadow_param .

proc p input n u32 . output u32 . effects none .
do
  let n u32 1 .     rem 매개변수 `n` 을 가린다
  return n. .
end .
```

진단: `E-NAME-SHADOW`

거부되는 예제 (rejected) — 같은 블록에서 같은 이름을 두 번 선언할 수 없다

소스 코드 (source)

```
module ex_shadow_twice .

proc p output u32 . effects none .
do
  let a u32 1 .
  let a u32 2 .     rem 첫 번째를 대신하지 않는다
  return a. .
end .
```

진단: `E-NAME-SHADOW`

거부되는 예제 (rejected) — 안쪽 블록이 바깥 이름을 가릴 수 없다

소스 코드 (source)

```
module ex_shadow_inner .

proc p input n u32 . output u32 . effects none .
do
  let a u32 1 .
  guard gt n. 0 . else do
```

```
    let a u32 2 .   rem 바깥 `a` 가 아직 살아 있다
    return a. .
  end .
  return a. .
end .
```

진단: `E-NAME-SHADOW`

6 이름을 찾는 차례는 지역 → 모듈 → 가져온 모듈이다. 가져온 이름이 서로 부딪히면 어느 것인지 밝혀 적어야 한다.

6.4.9 계약의 등급

1 계약 절 하나에 등급 (grade) 을 붙여 **그 조건을 언제 무엇으로 다룰지**를 정할 수 있다 — `requires <등급> <조건> .` 처럼 절 낱말 바로 뒤에 온다. 등급은 다음 셋이 전부다.

> **등급** (grade)
>
> 계약 절 하나에 붙여 그 조건을 **언제 누가 책임지는지**를 정하는 표시. 번역할 때 처리기가 증명하거나(`static`), 돌 때 검사하거나(`debug`), 사람이 참이라고 약속한다(`assume`).

표 34 — 계약의 등급 — 닫힌 집합 셋

등급	언제 보나	뜻
`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. 처리기는 이것을 **사실로 쓰지 아니한다**

2 등급을 적지 아니하면 처리기가 정한다 — 증명할 수 있으면 증명하고, 못 하면 빌드 모드(§6.4.7)가 정한 대로 남긴다. 곧 등급은 **기본 처분을 뒤집는 표시**이지 계약을 쓰는 데 필요한 것이 아니다.

3 `assume` 한 조건은 **사실로 쓰이지 아니한다.** 처리기는 그것을 지키지도 않고, 그것에 기대어 다른 검사를 지우지도 않는다 — 곧 그 조건은 **읽는 사람에게 하는 말**이다.

4 이것이 규범인 까닭은 그 반대가 성립하지 아니하기 때문이다. 지켜지지 않는 조건을 사실로 삼으면, 그 조건이 거짓일 때 처리기는 **지우지 말았어야 할 검사를 지운 채** 옳다고 믿는다. 곧 계약을 사실로 쓰려면 **강제해야 하고**, 강제하지 않는 것은 사실이 아니다.

표 35 — 같은 조건, 다른 처분

적은 것	진입에서	뒤따르는 검사
`requires le a 200`	검사한다	**지워진다** — 사실이 되었으므로
`requires assume le a 200`	검사하지 아니한다	남는다 — 사실이 아니므로

> **쉬운 말로**
>
> 그러면 `assume` 은 무엇에 쓰는가. 아직 처리기가 증명할 수 없지만 사람은 아는 것을 **적어 두는** 자리다. 적어 두면 읽는 사람이 알고, 나중에 처리기가 자라면 그 자리를 `static` 으로 올릴 수 있다. 적지 아니하면 그 앎은 사람의 머릿속에만 남는다.

> **예제 (example) — 같은 프로그램이 모드에 따라 다르게 끝난다 — `debug`**
>
> **소스 코드 (source)**

```
module ex_mode_debug .

build debug .

fn bump input a u8 . output u8 .
  requires le a. 200 . .
do
  return add a. 1 . .
end .
```

실행 결과 (output)

```
bump(10) = 11 · bump(250) → E-VM-CONTRACT (트랩)
```

예제 (example) — 같은 프로그램 — `release_fast`

소스 코드 (source)

```
module ex_mode_fast .

build release_fast .

fn bump input a u8 . output u8 .
  requires le a. 200 . .
do
  return add a. 1 . .
end .
```

실행 결과 (output)

```
bump(10) = 11 · bump(250) = 251 (멈추지 않는다)
```

예제 (example) — 같은 모드 — 계약이 넘침을 막던 자리

소스 코드 (source)

```
module ex_mode_fact .

build release_fast .

fn bump input a u8 . output u8 .
  requires le a. 254 . .
do
  return add a. 1 . .
end .
```

실행 결과 (output)

```
bump(10) = 11 · bump(255) → E-VM-OVERFLOW (트랩)
```

주의 — `release_fast` 는 안전한 부분집합 밖이다

같은 프로그램이 모드에 따라 다르게 끝난다. 계약 `requires le a. 200 . .` 을 어기고 `a` 를 250 으로 부르면 `debug` 에서는 멈추고(`E-VM-CONTRACT`), `release_fast` 에서는 **멈추지 않고 251 을 낸다.** 셋째 예제에서 계약은 `add a. 1 .` 이 넘치지 않음을 증명하던 근거였다. 모드가 계약을 없애면 그 근거도 없어지므로 넘침 검사가 돌아오고, `bump(255)` 는 256 을 내지 않고 멈춘다(§6.4.7 (4a)). 이것이 이 문서가 미정의 동작을 없앴다고 말하는 범위를 **안전한 부분집합**으로 한정하는 까닭 가운데 하나다(§4.4). 속도를 위해 검사를 없애는 것은 고

> 를 수 있는 일이되, **고른다는 사실이 소스에 남아야** 한다 — 그래서 모드는 명령줄 깃발이 아니라 `build` 문장이다.

6.4.10 지금 판정할 수 있는 위반

1 부르는 자리가 상대의 `requires` 를 어기고 **양쪽이 모두 상수**이면, 그 어김은 실행을 기다릴 필요 없이 **번역할 때** 판정된다. 이때 거부된다 (`E-CONTRACT-IMPOSSIBLE`).

2 이것은 계약을 더 엄하게 만든 것이 아니라, **언제 답이 나오는가**를 앞당긴 것이다. 전에는 초록으로 번역되어 실행 중에 덫에 걸렸다.

> **쉬운 말로**
>
> 번역할 때 답이 나는 계약이 실행할 때까지 미뤄지면, 그 프로그램은 **돌려 봐야만 틀렸음을 아는** 프로그램이 된다. 맞지 않는 비트 예산을 지닌 채 돌아가는 프로그램이 배포되는 일이 실제로 있었다. 답이 지금 나면 지금 말한다.

> **거부되는 예제 (rejected) — 양쪽이 상수이므로 지금 판정된다**
>
> **소스 코드 (source)**
>
> ```
> module ex_impossible .
>
> fn g input a u8 . output u8 . requires lt a. 10 . .
> do
>   return a. .
> end .
>
> fn f output u8 .
> do
>   return g 200 . .       rem 200 은 결코 10 보다 작지 않다
> end .
> ```
>
> 진단: `E-CONTRACT-IMPOSSIBLE`

6.4.11 이름이 될 수 없는 것

1 가릴 수 없는 것(§6.4.8)과는 별개로, **애초에 이름이 될 수 없는 글자**들이 있다.

표 36 — 이름으로 쓸 수 없는 것

무엇	진단	왜
열쇠말	`E-NAME-KEYWORD`	열쇠말은 읽는 사람이 **한눈에 짜임과 값을 가르는** 표시다
기본 연산의 이름	`E-NAME-BUILTIN`	이름칸이 **평평하므로**(가림 없음) 그 이름이 두 가지를 뜻하게 된다
_ 하나	`E-NAME-WILDCARD`	혼자 있는 _ 는 **아무거나**를 뜻하는 표시이지 이름이 아니다
__ 로 시작하는 이름	`E-NAME-VENDOR-RESERVED`	처리기와 라이브러리를 만드는 쪽의 몫으로 **남겨 둔** 자리다
ASCII 밖의 글자	`E-NAME-ASCII`	글월·문자열·`rem` 주석은 어떤 글자든 담지만, **이름은 ASCII** 다

1a 여기서 「기본 연산」은 §6.3 이 이름 부른 것들만이 아니다. **맨 이름으로 부르는 것**이면 그 집합에 든다 — 부록 D 의 기본 연산 표가 그 전부다. `call_builtin` 뒤에 서는 빌트인 연산(파일 · 그물 · 프로세스 · 표준 입출력 · 시계 · 난수 · 환경 · 터미널 · 부동소수점 함수 …)의 이름은 들지 아니한다 — 제 이름공간에 있다(§6.3.3 (1f)). **기본 연산의 이름이 잡혀 있다는 사실은 규범이다** — 지역 이름으로 쓰면 거부된다.

> **쉬운 말로**
>
> 목록을 적지 않으면서 「그 목록에 든 이름은 쓸 수 없다」고 정하는 것이 이상해 보일 수 있다. 그러나 두 물음은 다르다 — **무엇이 있는가**는 세는 쪽(원장)이 답하고, **그것이 이름칸을 차지하는가**는 규범이 답한다. 앞엣것은 늘고 줄지만 뒤엣것은 늘 참이며, **늘 참인 것만 이 문서에 적힌다.**

2 들여온 이름 둘이 **같은 이름에 묶이는** 것도 거부된다(`E-NAME-COLLISION`). 들여오기는 이미 있는 이름을 **덮어쓰지 아니한다.**

3 점이 든 이름 `<앞>.<뒤>` 에서 앞은 **선언된 타입**이어야 한다(`E-NAME-QUALIFIER`). 그 모양은 op 을 그 타입의 이름칸에 두는 것이지 모듈을 가리키는 것이 아니다.

> **쉬운 말로**
>
> 왜 ASCII 인가. 이 언어는 글월과 주석에 어떤 글자든 담을 수 있게 하면서 **이름만** 좁게 둔다. 이름은 사람이 읽을 뿐 아니라 도구가 옮기고 붙이고 비교하는 것이며, 눈으로 구별되지 않는 글자가 서로 다른 이름이 되는 일은 **찾을 수 없는 잘못**을 만든다.

> **주의 —**
>
> `__` 를 남겨 두는 것이 지키는 것은 처리기가 아니라 **쓰는 사람**이다. 그 자리가 늘 「안쪽의 것」을 뜻한다고 정해 두면, 남의 코드를 읽을 때 그것이 내 것이 아님을 이름만 보고 안다.

6.4.12 계약이 가리키는 것

1 계약 절이 부르는 이름은 그 자리에서 **볼 수 있는 것**이어야 한다. 없는 이름을 부르면 거부된다(`E-REQ-UNDEF` · `E-ENS-UNDEF`).

2 처리기가 그 자리로 **가져올 수 없는** 것을 부르는 계약도 거부된다 (`E-REQ-UNSUP` · `E-ENS-UNSUP`). 판단할 수 없는 것을 판단한 척하지 아니한다.

3 이름 붙인 계약(§6.4.5)이 선언되지 않은 것을 가리키면 거부된다 (`E-CONTRACT-UNDEF`).

4 일어날 수 **없는** 오류를 선언하는 것은 거부된다(`E-CONTRACT-DEAD`) — `requires` 가 이미 그 경우를 걸러 내고 있다면, 그 오류 갈래는 결코 나지 아니한다.

4a 같은 입력에 대한 `requires` 둘이 **함께 참일 수 없으면** 거부된다 (`E-CONTRACT-UNSAT`) — 어떤 인자로 불러도 진입에서 멈추므로 그 op 의 몸은 결코 돌지 아니한다. (4) 와 같은 자리다: 돌지 않는 코드가 시그니처에 적혀 있는 것이다.

> **쉬운 말로**
>
> (4) 가 막는 것은 틀린 프로그램이 아니라 **틀린 그림**이다. 결코 나지 않는 실패가 시그니처에 적혀 있으면, 부르는 쪽은 그것을 다루는 코드를 쓰고 그 코드는 영영 돌지 않는다. 돌지 않는 코드는 시험되지 않고, 시험되지 않는 코드는 언젠가 틀린다.

4b 처리기가 **강제하지 못하는** 계약 절은 그 사실을 알린다(`W-CONTRACT-IGNORED`). 진입 검사(`requires`)와 출구 검사(`ensures`)가 각각 아는 모양이 있고, 그 밖의 모양은 실행 중에 아무도 막지 않으며 구간 분석도 배우지 못한다. 그 절은 문서에만 선다.

> **쉬운 말로**
>
> (4b) 는 (2) 와 짝이다. (2) 는 **가져올 수 없는 것**을 거부하고, (4b) 는 가져올 수는 있으나 **세울 수 없는 모양**을 알린다. 둘 다 같은 하나를 지킨다 — 검사되지 않는 약속을 검사되는 것처럼 보이게 두지 아니한다. `ensures` 쪽은 2026-09-16 까지 이 알림이 없어 **조용히 버려졌다**(결함 노트 11).

5 번역 시점의 매개변수에 건넨 타입이 그 자리가 **요구하는 것을 갖추지 못하면** 거부된다(E-BOUND-UNSAT, §6.11.2).

6.4.13 시험 블록

> **시험 블록** (test block)
>
> `test <이름> do … end .` 로 적는, 입력도 출력도 없는 몸. 시험을 돌리라는 요청을 받았을 때만 실행된다.

> **단언** (assertion)
>
> `expect <조건> .` — 시험이 참이라고 주장하는 조건. 거짓이면 그 시험은 실패한다.

1 시험 블록은 모듈의 최상위에 온다. 머리에는 이름과, 있으면 `schedule` 절(§6.4.1 (3d))이 오고, 몸에는 op 의 몸에 오는 폼이 온다. 몸은 op 의 몸과 **같은 타입 규칙**으로 검사된다 — 시험 안이라고 `let x u8 300 .` 이 통과하지 아니한다.

1a 시험의 이름은 op · 타입 · 모듈의 이름과 **같은 이름 공간**에 든다. 한 모듈 안에서 시험끼리, 또는 시험과 다른 선언이 같은 이름을 쓰면 거부된다(E-NAME-DUP, §6.4.8 (2)). 시험은 이름으로 보고되고 이름으로 불리므로, 이름이 겹치면 어느 것을 가리키는지 말할 수 없다.

2 `expect <조건> .` 의 조건은 `bool` 이어야 한다. 수를 주는 것은 거부된다(E-TYPE-COND) — 0 이 아닌 수를 참으로 읽는 규칙은 이 언어에 없다(§6.2.16 (4)).

2a 시험을 돌렸을 때 `expect` 의 조건이 거짓이면 그 시험은 실패한다(E-TEST-FAIL). 이것은 계약 위반이 아니다 — 계약 위반은 코드가 **자기 약속**을 어긴 것이고, 시험 실패는 **적은 사람의 주장**과 코드가 다르다는 것이다. 고칠 자리가 다르다.

2b 처리기는 어느 빌드 모드(§6.4.7)에서도 `expect` 를 없애지 아니한다. 사라진 단언은 돌지 않은 시험이다.

2c `expect` 는 시험 블록 안에만 온다. op 의 몸이나 그 밖의 자리에 적으면 거부된다(E-EXPECT-PLACE). 입력이나 결과에 대한 약속은 `requires` · `ensures`(§6.4.3)로, 일부러 멈추는 것은 `panic`(§7.1)으로 적는다.

3 시험은 시험을 돌리라는 요청(`lowentc --test`)을 받았을 때만 실행된다. 보통의 실행과 검사(`--check`)는 시험을 돌리지 아니하며, 검사는 그 사실을 알린다(W-TEST-NOT-RUN) — 검사의 통과가 시험의 통과로 읽히지 않게 하기 위함이다.

3a 번역이 거부하는 단위의 시험은 돌지 아니한다. 시험을 돌리는 요청도 먼저 `--check` 와 같은 검사를 거친다. 거부된 프로그램이 «통과» 를 내면 그 초록은 거짓이다.

3b 시험을 돌린 결과는 돈 시험의 수 · 통과한 수 · 실패한 수다. 시험이 하나도 없으면 없다고 말한다. 실패한 시험이 하나라도 있으면 그 요청은 실패로 끝난다.

4 `schedule explore_interleavings [limit <수>] .` 절이 붙은 시험은 메시지가 배달되는 **차례**를 바꾸어 가며 되풀이해 돌린다. 바꾸는 것은 §10.5 (4b) 가 열어 둔 차례뿐이다 — 한 액터 안의 차례는 하나다(§10.5 (4a)). `limit` 이 없으면 처리기가 정한 상한까지 돈다. 상한 전에 차례를 다 돌면 결과를 «전수» 로, 상한에서 멈추면 «상한까지» 로 보고한다. 상한에서 멈춘 것을 전수라고 보고하지 아니한다.

4a 모든 차례에서 통과하면 그 시험은 통과한다. 어떤 차례에서는 통과하고 어떤 차례에서는 실패하면 그 시험은 실패한다(E-SCHED-NONDET) — 결과가 배달 차례에 기대는 것이고, 한 번 돌려 본 답은 그 한 차례의 답일 뿐이다. 모든 차례에서 실패하면 보통의 시험 실패(E-TEST-FAIL)다.

예제 (example) — 시험 블록과 단언

소스 코드 (source)

```
module ex_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 .
```

거부되는 예제 (rejected) — 단언의 조건은 bool 이다 — 수는 참이 아니다

소스 코드 (source)

```
module ex_test_num .

test counts do
  expect 1 .
end .
```

진단: `E-TYPE-COND`

거부되는 예제 (rejected) — 한 모듈의 두 시험은 이름이 달라야 한다

소스 코드 (source)

```
module ex_test_dup .

test same do
  expect eq 1 1 . .
end .

test same do
  expect eq 2 2 . .
end .
```

진단: `E-NAME-DUP`

거부되는 예제 (rejected) — 시험과 op 은 한 이름 공간을 쓴다

소스 코드 (source)

```
module ex_test_ns .

fn check output bool . do return true . end .

test check do
  expect check. .
end .
```

진단: `E-NAME-DUP`

거부되는 예제 (rejected) — expect 는 시험 블록 안에만 온다

소스 코드 (source)

```
module ex_expect_place .

fn half input n u8 . output u8 .
do
  expect lt n. 200 . .
  return div n. 2 . .
end .
```

진단: `E-EXPECT-PLACE`

6.5 문장과 제어 (Statements and control flow)

1 문장 (statement) 은 실행되는 것이다. 식은 값을 만들고, 문장은 일을 한다.

1a 그러므로 **값만 내고 아무것도 바꾸지 않는** 폼 — 순수한 셈(`add s 100 . · eq s 7 .`)이나 `fn` 부름 — 을 문장으로 쓰는 것은 적합하지 아니하다(`E-VALUE-DISCARDED`). 그 값은 아무 데도 가지 않는다. 대개 뜻한 것은 `set s. add s. 100 . .` 이다. `proc` 부름은 효과가 일이므로 문장이 된다.

거부되는 예제 (rejected) — 값만 내는 폼을 문장으로 쓸 수 없다

소스 코드 (source)

```
module ex_discard .

fn bump input a u64 . output u64 .
  requires lt a. 100 . .
do
  var s u64 a. .
  add s. 100 .   rem 뜻한 것은 set s (add s 100) . 이다
  return s. .
end .
```

진단: `E-VALUE-DISCARDED`

6.5.1 이름을 짓는 문장

1 `let` 은 **바뀌지 않는** 이름을 짓는다. 한 번 정해진 값은 바뀌지 아니한다.

2 `var` 는 **바뀔 수 있는** 이름을 짓는다. `set` 으로 값을 바꾼다.

2a **매개변수도 `set` 할 수 있다.** 값으로 받은 매개변수는 이 op 안의 **지역 복사**이므로, 그것을 바꾸어도 부른 쪽의 값은 바뀌지 아니한다. struct 매개변수의 칸(배열 칸 포함)에 쓰는 것도 같다 — 지역 복사에 쓴다. 부른 쪽에 닿는 쓰기는 `mut·mut_ref` 자리뿐이다(§8.4).

2b 그러므로 `requires·ensures` 가 매개변수 이름으로 말하는 것은 **들어올 때의 값**이다. 몸통이 그 이름을 `set` 한 뒤에도 계약이 말하는 것은 들어올 때의 값 그대로다.

```
binding      ::= ( "let" ¦ "var" ) name [ alloc-clause ] [ type ] term [ fail-clause ] "."
alloc-clause ::= ( "use" ¦ "keep" ) name "."
fail-clause  ::= "else" leave "."
               ¦ "else" block
               ¦ "else" "error" name block
leave        ::= "return" [ term ] ¦ "break" [ number ] ¦ "continue" [ number ] ¦ "panic" term
```

3 이름을 짓는 문장은 `let <이름> <타입> <값> .`(`var` 도 같다)이다 — **타입은 이름 뒤, 값 앞에 선다.** 값이 그 타입에 맞지 않으면 번역이 거부된다. 타입은 추측하지 아니한다: `let x u64 300 .` 이지 `let x 300 .` 이 아니다(`E-LET-NOTYPE`). 추측한 타입은 맨 리터럴이 폭 검사를 빠져나가게 했다. 값이 `lit` 로 시작하면 타입은 그 `lit` 뒤에 적혀 있으므로 앞에 다시 적지 아니할 수 있다(`let p lit pt do … end . .`).

3a 이름과 타입 사이에 낱말을 끼우지 아니한다. 옛 꼴 `let <이름> be <타입> <값> .` 의 `be` 는 이 언어의 낱말이 아니다 — 적으면 번역이 거부된다(E-LET-BE).

3b 타입은 글자만으로 끝난다: 닫힌 어휘의 생성자(`u64` · `slice u8` · `result u64 perr` · `array 4 u8`)이거나, 이름 하나이거나, 괄호다. 인자를 받는 사용자 타입과 인자 수가 정해지지 않은 형태(오류 타입을 적지 않은 `result u64` 따위)는 괄호로 싼다 — `let b (box u64) mk. .` · `let r (result u64) ok k. . ..`

3c **할당기 절**은 이름 바로 뒤, 타입 앞에 선다 — `use <할당기>.` 는 블록이 끝날 때 돌려주고, `keep <할당기>.` 는 블록을 나가도 돌려주지 아니한다(§6.2.6 (7e)(7f) · §8.14). 할당기는 변수이므로 점으로 닫는다: `var v use al. option (vec u32 allocs.bump_bytes) vecgen.new al. 8 . ..` 옛 꼴 `using <할당기> be` · `using <할당기> keep be` 는 거부된다(E-USING-OLD). 타입에 `owned` 가 든 바인딩 (`let n use al. option (owned node) lit node … .`)은 `use` 로 적어도 블록 끝에 돌려주지 아니한다 — §8.5 (10)(11).

4 타입 뒤에는 **값이 있어야 한다.** 값 없이 닫으면 번역이 거부된다(E-LET-NOVALUE). 이름을 짓되 값을 나중에 주는 길은 이 언어에 없다.

4a 값은 폼 하나이고 **제 점으로 닫힌다**(§6.1.5). 그 뒤에 문장의 닫는 점이 오거나 `else` 가 온다 — `let n u64 find k. . .` · `let n u64 find k. . else return 0 . ..` 실패 절은 문장의 닫는 점 **앞에** 선다: 문장을 닫은 뒤에 `else` 를 적으면 번역이 거부된다(E-STMT-ELSE).

5 **실패 절** — `let <이름> <타입> <값> else <벗어남> . .` 에서 값의 타입이 `option t`·`result t e` 이고 적은 타입이 알맹이 `t` 이면, 값이 있을 때 그 알맹이가 이름에 묶이고 비었을 때(none·오류) `else` 뒤가 실행된다. `else` 뒤는 **반드시 벗어나야 한다**(return·break·continue·panic) — 흘러 내려오면 거부된다 (E-GUARD-FALLTHROUGH). 그래서 그 뒤에서 이름은 늘 알맹이를 가진다. `var` 도 같다.

5c 실패 절의 꼴은 셋이다. `else <벗어남> .` 은 벗어나는 문장 하나다 — 그 문장은 제 점으로 닫힌다. `else do … end` 는 여러 문장이다 — 블록의 모든 길이 벗어나야 한다. `else error <이름> do … end` 는 `result` 의 **오류 값을 `<이름>` 에 묶어** 블록 안에서 본다 — `<이름>` 은 그 블록 안에서만 살고, 블록의 모든 길이 벗어나야 한다. 값이 `option` 이면 받을 오류가 없으므로 `else error <이름>` 은 거부된다(E-BIND-ELSE).

5d 실패 절 뒤에는 **문장의 닫는 점**이 온다 — 벗어나는 문장 뒤에도(`… else return 0 . .`), 블록 뒤에도 (`… else do … end .`). 그 점이 빠지면 번역이 거부된다(E-DOT-MISSING).

5a 값이 `option`·`result` 가 아니거나 알맹이의 타입이 적은 타입과 다르면 거부된다(E-BIND-ELSE). 값의 타입은 부르는 op 의 출력 · 이름의 선언 · actor 처리기의 출력에서 읽는다 — 그 밖의 식이면 먼저 타입을 적어 묶는다.

5b `else` 없이 `option`·`result` 값을 알맹이 타입으로 묶으면 거부된다(E-TYPE-LET) — 실패할 수 있음을 잊은 자리다. 멈춰도 되면 `else panic "…" .` 으로 적는다(효과 `panic` 이 따른다).

> **주의 —**
>
> 이 규칙이 잡는 것은 빈칸을 적는 실수만이 아니다. `let x f64 .5 .` 이라고 적으면 `.5` 는 부동소수점 리터럴이 아니므로(§6.1.4 (2)) 그 점이 폼을 닫고, 이름 뒤에는 **타입만 남는다.** 사람은 값을 적었다고 믿는데 처리기는 값을 못 본 자리이며, 그래서 진단이 그 함정을 이름으로 짚는다.

> **참고 (informative)**
>
> 기본이 `let` 인 것이 중요하다. 바뀌지 않는 이름은 읽는 사람이 **한 번만 확인하면** 되지만, 바뀔 수 있는 이름은 쓰이는 자리마다 "여기서는 무슨 값이지"를 다시 물어야 한다. 그 물음이 곧 의미 엔트로피다(§1.3).

> **거부되는 예제 (rejected) — 타입 뒤에 값이 없다**
>
> **소스 코드 (source)**
>
> ```
> module ex_let_novalue .
>
> fn f output u8 .
> do
> ```

```
  let a u8 .        rem 조용히 0 을 넣지 아니한다
  return a. .
end .
```

진단: `E-LET-NOVALUE`

거부되는 예제 (rejected) — 타입을 추측하지 않는다

소스 코드 (source)

```
module ex_let_notype .

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

진단: `E-LET-NOTYPE`

거부되는 예제 (rejected) — `be` 는 이 언어의 낱말이 아니다 — (3a) 를 시험한다

소스 코드 (source)

```
module ex_let_be .

fn f output u64 .
do
  let x be u64 7 . .
  return x. .
end .
```

진단: `E-LET-BE`

거부되는 예제 (rejected) — 실패 절은 문장의 닫는 점 앞에 온다 — (4a) 를 시험한다

소스 코드 (source)

```
module ex_else_dot .

fn find input k u64 . output option u64 .
do
  if eq k. 0 . do return none . end .
  return some k. . .
end .

fn f input k u64 . output u64 .
do
  let n u64 find k. . . else return 0 .
  return n. .
end .
```

진단: `E-STMT-ELSE`

거부되는 예제 (rejected) — 할당기 절은 `use` · `keep` 이다 — (3c) 를 시험한다

소스 코드 (source)

```
module ex_using_old .
```

```
actor exact do
  state do n u64 . end .
  proc reserve input k u64 . output option mut slice u8 . effects state . do
    return none .
  end .
end .

proc f output u64 . effects state .
do
  var e exact spawn actor exact . .
  let b using e. mut slice u8 lit array 4 u8 _ . else return 0 . .
  return len b. . .
end .
```

진단: `E-USING-OLD`

예제 (example) — 실패 절 셋 — (5)(5c) 를 시험한다

소스 코드 (source)

```
module ex_fail_clause .

enum perr do bad . empty . end .

fn parse input s u64 . output result u64 perr .
  errors empty eq s. 0 . .
  errors bad gt s. 100 . .
do
  if eq s. 0 . do return error empty . . end .
  if gt s. 100 . do return error bad . . end .
  return ok s. . .
end .

export fn one input s u64 . output u64 .
do
  let n u64 parse s. . else return 0 . .
  return add n. 1 . .
end .

export fn many input s u64 . output u64 .
do
  let n u64 parse s. . else do
    return 99 .
  end .
  return add n. 1 . .
end .

export fn why input s u64 . output u64 .
do
  let n u64 parse s. . else error e do
    if eq e. bad. . do return 1000 . end .
    return 2000 .
  end .
  return add n. 1 . .
end .
```

실행 결과 (output)

```
one(5) = 6 · one(0) = 0 · many(200) = 99 · why(0) = 2000 · why(500) = 1000
```

6.5.2 조건 — `if`

구문

```
if-statement ::= "if" term block { "else" "if" term block } [ "else" block ] "."
block        ::= "do" { statement } "end"
```

제약

조건(`term`)의 타입은 `bool` 이어야 한다(§6.2.3).

`if-statement` 는 **문**이다. 값이 놓이는 자리에 올 수 없다.

동적 의미

조건을 차례로 계산한다.

그 값이 참이면 첫 `block` 의 문장을 적힌 차례로 실행한다.

그 값이 거짓이고 `else if` 가 있으면 그 조건으로 (3) 부터 다시 하고, `else` 가 있으면 그 뒤의 `block` 을 실행한다.

그 값이 거짓이고 `else` 가 없으면 아무것도 실행하지 아니한다.

진단

표 37 — if 문의 진단

어긴 것	진단
(1) — 조건이 `bool` 이 아니다	`E-TYPE-COND`
(2) — `if` 를 값 자리에 적었다	`E-IF-VALUE`
구문 — 조건이 닫힌 뒤에 점을 하나 더 적었다(`if c. . do`)	`E-CLOSER-EXTRA`
구문 — 사슬 끝의 닫는 점이 없다	`E-DOT-MISSING`
구문 — `block` 자리에 `do … end` 가 아닌 것을 적었다	`E-CTRL-NODO`
구문 — `else` 를 `block` 안에 적었다	`E-STMT-ELSE`

예제

예제 (example) — 갈래 셋 — (4)(5) 를 시험한다

소스 코드 (source)

```
module ex_if .

export fn sign input a i64 . output u64 .
do
  if gt a. 0 . do
    return 1 .
  end else if eq a. 0 . do
    return 0 .
  end else do
    return 2 .
  end .
end .
```

실행 결과 (output)

```
sign(5) = 1 · sign(0) = 0 · sign(-3) = 2
```

거부되는 예제 (rejected) — `if` 는 값을 내지 아니한다 — (2) 를 시험한다

소스 코드 (source)

```
module ex_if_value .

fn pick input a u64 . output u64 .
do
  let x u64 (if gt a 1 do 5 . end else do 6 . end) .   rem 갈래마다 set 하거나 return 한다
  return x. .
end .
```

진단: `E-IF-VALUE`

참고

쉬운 말로

왜 몸이 블록이어야 하는가. 조건은 제 점에서 끝난다(`if gt a. 3 . do`) — 조건이 다음 낱말을 삼킬 일은 없다. 그래도 몸을 문장 하나로 적게 두면 같은 `if` 가 두 모양이 되고, 둘째 문장을 덧붙이는 순간 그것이 `if` 안인지 밖인지가 들여쓰기에 달린다. 몸이 늘 `do … end` 이므로 몸의 끝은 `end` 가, 문장의 끝은 그 뒤의 점이 말한다.

참고 (informative)

머리의 식이 `do` 에서 끝나고 몸을 블록으로 적는 것은 `while`(§6.5.3) · 되풀이의 네 머리(§6.5.3.1) · `match`(§6.6) 도 같다. 머리의 식이 블록을 품은 값이면(`if eq p. lit pt do x 1 . end . . do`) 머리의 몸은 **마지막 블록**이다 — 읽는 사람을 위해 그런 값은 괄호로 싼다.

거부되는 예제 (rejected) — 머리의 식 뒤에 점이 하나 더 있다

소스 코드 (source)

```
module ex_ctrl_dot .

fn f input a u64 . output u64 .
do
  if gt a. 3 . . do
    return 1 .
  end .
  return 0 .
end .
```

진단: `E-CLOSER-EXTRA`

참고 (informative)

갈래마다 다른 값을 얻으려면 각 갈래에서 이름에 `set` 하거나 `return` 한다.

6.5.3 되풀이 — `while`

1 `while <조건> do <문장들> end .` 는 조건이 참인 동안 블록을 되풀이한다. 조건은 **바퀴마다, 블록에 들어가기 전에** 계산한다 — 처음부터 거짓이면 블록은 한 번도 돌지 아니한다.

1a 조건은 폼 하나이고 제 점으로 닫힌다(`while lt i. n. . do … end .`). 그 뒤에 점을 하나 더 적으면 번역이 거부된다 (`E-CLOSER-EXTRA`). 문장은 `end` 다음의 점에서 끝난다(§6.1.6 (2b)).

2 `break` 는 되풀이를 벗어나고, `continue` 는 다음 바퀴로 넘어간다. 수를 적지 않으면 둘은 **가장 안쪽** 되풀이에 듣는다. `break` 는 값을 나르지 아니한다.

2b 낱말 뒤에 수를 적으면 그 수만큼 **바깥으로 센** 되풀이에 듣는다. `break 2 .` 는 가장 안쪽 되풀이와 그것을 감싼 되풀이를 함께 벗어난다. `continue 2 .` 는 가장 안쪽 되풀이를 벗어나, 그것을 감싼 되풀이의 다음 바퀴로 간다. `1` 은 수를 적지 않은 것과 같다. 수는 꾸밈 없는 십진 리터럴이며, `1` 이상이고 그 문장을 감싼 되풀이의 수를 넘지 못한다(`E-LOOP-COUNT`). 되풀이에 붙이는 이름(라벨)은 없다.

2c 세는 것은 되풀이의 다섯 머리(`while` · `for` · `repeat` · `range` · `cycle`)다. `if` 와 `match` 의 블록은 세지 아니한다. 벗어나는 길에서 지나가는 되풀이가 연 것은 그 되풀이를 여느 때 벗어날 때와 같이 닫힌다.

> **참고 (informative)**
>
> 되풀이를 한 겹 감싸거나 풀면 같은 수가 다른 되풀이를 가리킨다. 그때는 수를 다시 센다.

2a 그 둘은 **되풀이 안에서만** 쓴다. 되풀이 밖에 적는 것은 적합하지 아니하다 (`E-LOOP-OUTSIDE`) — 벗어날 되풀이가 없다.

3 되풀이의 조건도 `bool` 이어야 한다. 수를 조건 자리에 적으면 번역이 거부된다(`E-TYPE-COND`).

3a `while` 은 문장이며 값을 내지 아니한다. `else` 갈래를 갖지 아니한다(`E-STMT-ELSE`).

> **예제 (example) — 겹친 되풀이를 한 번에 벗어난다**
>
> **소스 코드 (source)**
>
> ```
> module ex_break_count .
>
> export fn first_zero input want u32 . output usize .
> do
>   let g lit slice array 3 u32 lit array 3 u32 4 7 1 . lit array 3 u32 9 0 5 . . .
>   var hit usize 99 .
>   repeat r usize len g. . do
>     repeat c usize 3 do
>       if eq idx idx g. r. . c. . want. . do
>         set hit. add mul r. 3 . c. . .
>         break 2 .
>       end .
>     end .
>   end .
>   return hit. .
> end .
> ```

> **거부되는 예제 (rejected) — 감싼 되풀이는 둘인데 셋을 벗어나려 한다**
>
> **소스 코드 (source)**
>
> ```
> module ex_break_far .
>
> export fn f output u64 . do
>   var n u64 0 .
>   repeat a u64 3 do
>     repeat b u64 3 do
>       set n. add n. 1 . .
>       break 3 .
>     end .
>   end .
>   return n. .
> end .
> ```
>
> 진단: `E-LOOP-COUNT`

예제 (example) — 조건과 되풀이

소스 코드 (source)

```
module ex_ctl .

export fn count_big input n u32 . output u32 .
  requires le n. 100 . .
do
  var total u32 0 .
  var i u32 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 .
```

6.5.3.1 차례로 도는 되풀이 — `for` · `repeat` · `range` · `cycle`

```
for-stmt    ::= "for"    name [ "mut" ] term [ filter ] block "."
repeat-stmt ::= "repeat" name type term [ filter ] block "."
range-stmt  ::= "range"  name type term term [ "step" term ] [ filter ] block "."
cycle-stmt  ::= "cycle"  name type term "while" term "next" term [ filter ] block "."
filter      ::= "if" term
```

1 네 문장은 머리가 정한 값을 **차례로 이름에 담아** 블록을 되풀이한다. 머리말마다 하는 일이 하나다 — `for` 는 원천의 원소를, `repeat` 는 0 부터 n 번을, `range` 는 두 끝 사이를, `cycle` 은 저자가 적은 다음 값을 돈다. 그 밖의 꼴은 없다.

표 38 — 되풀이의 머리

머리	꼴	이름이 차례로 갖는 것
원소	`for x <슬라이스> do`	슬라이스의 원소 — 앞에서부터
원소 자리	`for x mut <슬라이스> do`	슬라이스의 칸 그 자체 — 읽고 쓸 수 있다
세기	`repeat i <타입> n do`	`0, 1, …, n-1`
구간	`range i <타입> a b do` · `range i <타입> a b step k do`	`a` 에서 `b` 까지, 두 끝을 넣어
점화식	`cycle i <타입> v while c next e do`	`v` 에서 시작해 바퀴마다 `e`

1a 머리의 식은 저마다 폼 하나이고 제 점으로 닫힌다 — `repeat i u64 len xs. . do` · `range i u64 0 sub n. 1 . do` · `cycle i u64 1 while lt i. n. . next mul i. 2 . do`. 끝내는 낱말(`do` · `if` · `while` · `next` · `step`)은 예약어이므로 식 안에 이름으로 올 수 없다. 닫힌 식 뒤에 점을 하나 더 적으면 번역이 거부된다(E-CLOSER-EXTRA).

1b 옛 꼴 `for i count <타입> n . do` · `for i range <타입> a b . do` · `for i be <타입> v . while c . next e . do` 와 거르기 `. where c .` 는 거부된다(E-FOR-OLD). `for` 는 원소를 도는 한 가지 일만 한다.

2 **원소.** 원천은 슬라이스여야 한다. 슬라이스가 아니면 번역이 거부된다(E-TYPE-ITER). 이름의 타입은 슬라이스의 **원소 타입**이다. 문자열 리터럴(바이트의 슬라이스)과 나열 리터럴(`lit slice u16 10 20 30 .`)도 원천이다. 원소가 없으면 블록은 한 번도 돌지 아니한다.

2a 원천은 `do`(또는 거르기의 `if`)에서 끝난다 — 이름 하나든(`for x xs. do`) 식이든(`for x view_array u16 b. . do`) 같다.

3 **원소 자리.** `for x mut buf. do … end .` 의 `x` 는 `buf` 의 그 칸이다: 읽으면 칸의 값이고, `set x. v. .` 는 그 칸에 쓴다. `buf` 는 `mut` 이어야 한다(E-TYPE-MUT). 되풀이 동안 `buf` 전체를 빌리므로 블록 안에서 `buf` 를 읽거나 쓰면 번역이 거부된다(E-FOR-HEAD) — 원소는 `x` 로만 만진다.

3a `mut` 없이 적은 `for x <슬라이스> do` 의 `x` 는 그 바퀴의 원소 값을 담은 **사본**이다. 블록 안에서 `x` 에 `set` 할 수 없다(E-FOR-HEAD) — 칸을 바꾸려면 머리에 `mut` 을 적는다.

4 **세기.** `repeat i <타입> n do` 는 `i` 에 `0` 부터 `n-1` 까지를 차례로 담는다. `n` 이 0 이하이면 한 번도 돌지 아니한다. `<타입>` 는 셈의 타입이며 **언제나 적는 정수 타입**이다 — 적지 않거나 정수 타입이 아니면 번역이 거부된다(E-FOR-STEP). `n` 은 `<타입>` 에 들어가야 한다(E-TYPE-WIDTH).

5 **구간.** `range i <타입> a b do` 는 `a` 에서 `b` 까지 **두 끝을 넣어** 돈다. 방향은 두 끝이 정한다 — `range i u64 3 5` 는 `3, 4, 5`, `range i u64 5 3` 은 `5, 4, 3` 이다. 두 끝이 같으면 한 번 돈다. 두 끝은 `<타입>` 에 들어가야 한다(E-TYPE-WIDTH).

5a `step k` 를 적으면 다음 값은 *지금* `+ k` 이고 **방향은 `k` 의 부호가 정한다.** 시작이 이미 그 방향으로 끝을 넘었으면 한 번도 돌지 아니한다(`range i i64 1 10 step -1`). 다음 값이 끝을 지나치면 거기서 끝난다 — `range i u8 250 255 step 3` 은 `250, 253` 이다. `step 0` 은 번역이 거부되고(E-FOR-STEP), 번역 때 알 수 없는 `k` 가 실행 중에 0 이면 멈춘다.

5b 셈은 **넘치지 아니한다.** `range i u8 0 255` 는 256 번 돌고 255 에서 끝난다 — 마지막 값 다음을 계산하다 타입을 넘는 일이 없다.

6 **점화식.** `cycle i <타입> v while c next e do … end .` 는 `i` 를 `v` 로 시작한다. 바퀴에 들어가기 전에 `c` 를 보아 참이면 블록을 돌고, 블록이 끝나면 `i` 에 `e` 의 값을 담는다. `c` 가 처음부터 거짓이면 한 번도 돌지 아니한다. `c` 는 `bool` 이어야 하고(E-TYPE-COND), `v` 와 `e` 는 `<타입>` 에 들어가야 한다(E-TYPE-WIDTH). `e` 의 계산은 보통의 산술이다 — 넘치면 멈춘다(§6.3.4). `while` · `next` 두 절이 이 차례로 모두 있어야 한다 (E-FOR-HEAD). 블록 안의 `continue` 는 `next` 를 **거친다** — `i` 에 `e` 의 값을 담고 `c` 를 다시 본다.

7 **거르기.** 머리 끝, `do` 앞에 `if c` 를 덧붙일 수 있다. 블록 첫머리에 `if not c. . do continue . end .` 를 적은 것과 같은 뜻이다 — 조건은 바퀴마다 계산한다. 네 머리말 어느 것에나 붙는다. 머리 안의 `if` 는 문장이 아니라 절이다 — 머리 안에는 문장이 올 수 없으므로 겹치지 아니한다.

8 **머리가 읽은 것은 되풀이 동안 얼린다.** 끝 · `step` · 원천은 되풀이에 들어갈 때 **한 번** 계산한다. 셈이름(`repeat` · `range` · `cycle` 의 이름)과 머리가 읽은 이름은 블록 안에서 `set` 할 수 없다 (E-FOR-HEAD). 슬라이스 칸의 내용을 바꾸는 것은 된다.

9 되풀이의 이름은 **블록 안에서만** 산다. 블록이 끝나면 그 이름은 없다 — 블록 뒤에서 읽으면 번역이 거부된다(E-NAME-SCOPE). 같은 블록에 이미 있는 이름을 다시 쓸 수 없다(E-NAME-SHADOW).

10 `break` 와 `continue` 는 네 되풀이 안에서도 `while` 에서와 같이 쓴다(§6.5.3 (2)). 넷 모두 값을 내지 아니한다.

> **예제 (example) — 수를 세는 머리**

소스 코드 (source)

```
module ex_for_count .

export fn evens output u64 .
do
  var acc u64 0 .
  range i u64 10 1 step -2 do
    set acc. add acc. i. . .
  end .
  return acc. .
end .
```

실행 결과 (output)

```
1 부터 10 까지 짝수의 합
```

예제 (example) — 슬라이스를 훑기

소스 코드 (source)

```
module ex_for .

export fn total_of input xs slice u8 . output u64 .
do
  var acc u64 0 .
  for x xs. do
    set acc. add acc. widen u64 x. . . .
  end .
  return acc. .
end .
```

실행 결과 (output)

```
원소를 모두 더한다
```

예제 (example) — 원소 자리 · 점화식 · 거르기

소스 코드 (source)

```
module ex_for_more .

fn bump input b mut slice u64 . output u64 .
do
  for x mut b. do
    set x. mul x. 2 . .
  end .
  var acc u64 0 .
  for y b. do
    set acc. add acc. y. . .
  end .
  return acc. .
end .

export fn powers output u64 .
do
  var n u64 0 .
  cycle i u64 1 while lt i. 100 . next mul i. 2 . do
    set n. add n. 1 . .
  end .
  return n. .
end .

export fn evens output u64 .
do
  var acc u64 0 .
  repeat i u64 10 if eq mod i. 2 . 0 . do
    set acc. add acc. i. . .
  end .
  return acc. .
end .
```

실행 결과 (output)

```
powers() = 7 · evens() = 20
```

거부되는 예제 (rejected) — 셈 이름은 블록 안에서 바꿀 수 없다

소스 코드 (source)

```
module ex_for_head .

fn f output u64 .
do
  repeat i u64 3 do
    set i. 0 .
  end .
  return 0 .
end .
```

진단: `E-FOR-HEAD`

거부되는 예제 (rejected) — `for` 는 원소만 돈다 — (1b) 를 시험한다

소스 코드 (source)

```
module ex_for_old .

fn f output u64 .
do
  var acc u64 0 .
  for i count u64 3 do
    set acc (add acc i) .
  end
  return acc. .
end .
```

진단: `E-FOR-OLD`

주의 —

`in` 은 이 언어의 낱말이 **아니다**. `for x in xs` 라고 적으면 거부된다 (`E-VOCAB-REMOVED`). 훑을 대상은 이름 바로 뒤에 온다.

6.5.4 빠져나가는 조건 — `guard`

1 `guard` 는 조건이 참이 아니면 **그 자리에서 빠져나간다**. `else` 뒤에 오는 것은 **모든 길이 빠져나가야** 한다.

0 `guard <조건> else <벗어남> . .` — 조건은 폼 하나이고 제 점으로 닫힌다. `else` 뒤의 벗어나는 문장도 제 점으로 닫히고, 그 뒤에 `guard` 를 닫는 점이 온다(`guard lt i. 9 . else return 0 . .`). `else` 뒤가 블록이어도 같다 (`guard c. else do … end .`).

1a `else` 뒤에는 한 문장이 올 수도 있고 블록이 올 수도 있다. 블록이면 그 블록의 **모든 길**이 빠져나가야 한다 — 빠져나가는 문장은 `return`, `break`, `continue`, `panic` 이다.

2 `else` 의 어떤 길이 빠져나가지 않고 아래로 이어지면 번역이 거부된다.

3 `guard` 를 지나면 그 조건은 **참임이 보장된다.** 처리기는 그 뒤의 코드에서 그 사실을 쓴다.

거부되는 예제 (rejected) — `else` 가 빠져나가지 않는다

소스 코드 (source)

```
module ex_guard_bad .

proc p input n u32 . output u32 . effects none .
do
  guard le n. 5 . else set n. 0 . .   rem 빠져나가지 않고 아래로 이어진다
  return n. .
end .
```

진단: `E-GUARD-FALLTHROUGH`

주의 — `guard` 는 `if not` 의 다른 이름이 아니다

`guard` 가 하는 말은 "이 조건이 아니면 여기서 끝"이다. 그래서 `guard` 아래의 코드는 조건이 참인 세계에서만 산다. `else` 가 빠져나가지 않으면 그 약속이 깨지고, **아래 코드가 참이라고 잘못 믿게 된다** — 그래서 언어가 거부한다.

예제 (example) — guard

소스 코드 (source)

```
module ex_guard .

export fn safe_head input data slice u8 . output u8 .
do
  guard ge len data. . 1 . else return 0 . .
  return idx data. 0 . .
end .
```

예제 (example) — 타입은 값 앞에 적고, `guard else` 는 블록이어도 된다

소스 코드 (source)

```
module ex_infer_guard .

rem 타입은 이름 뒤, 값 앞에 적는다 — 추측하지 않는다.
fn inferred output u64 .
do
  let a u64 7 .
  return a. .
end .

rem `else` 가 블록이어도 된다. 규칙은 "모든 길이 빠져나가는가" 다.
fn guarded input n u8 . output u8 .
do
  guard gt n. 5 . else do
    let x u8 1 .
    return x. .
  end .
  return 9 .
end .
```

실행 결과 (output)

```
inferred() = 7 · guarded(3) = 1 · guarded(9) = 9
```

거부되는 예제 (rejected) — `else` 의 길이 빠져나가지 않으면 거부된다

소스 코드 (source)

```
module ex_guard_fall .

fn f input n u8 . output u8 .
do
  guard gt n. 5 . else do
    let x u8 1 .
  end .
  return 9 .
end .
```

진단: `E-GUARD-FALLTHROUGH`

6.5.5 돌아가기 — `return`

1 `return` 은 op 을 끝내고 값을 돌려준다. 돌려주는 값의 타입은 `output` 에 적은 것과 같아야 한다.

1a 이 규칙은 `return` 이 **어디에 있든** 같다 — `guard` 의 `else` 뒤의 한 문장과 `else` 블록 안의 모든 문장도 같은 검사를 받는다. 그래서 `output result …` 인 op 은 `ok <값>` · `error <변형>` · `result` 를 내는 식으로만 돌아가고, 맨값을 돌려주면 거부된다 (`E-TYPE-RETURN`). 반대로 맨 타입을 적은 op 이 `ok <값>` 을 돌려주는 것도 거부된다.

거부되는 예제 (rejected) — `result` 자리에 맨값을 돌려준다

소스 코드 (source)

```
module ex_bare_under_result .

enum short do
  too_short .
end .

fn head input b slice u8 . output result u8 short .
errors too_short .
do
  guard ge len b. . 2 . else return 0 . .   rem `ok 0` 도 `error too_short` 도 아니다
  return ok idx b. 0 . . .
end .
```

진단: `E-TYPE-RETURN`

쉬운 말로

이 자리는 한동안 **검사 밖**이었다(2026-09-25 까지) — `guard` 의 `else` 는 문장 목록이 아니라 한 덩이로 붙어 있어서 검사기가 그 안으로 내려가지 않았다. 그래서 위의 op 은 번역을 통과했고, 부르는 쪽이 `is_ok` 를 묻는 순간 **실행 중에** 멈췄다. 드문 길(입력이 모자람)일수록 늦게 드러난다.

1b 출력에 이름을 붙인 op(§6.4.1 (3e))은 그 이름을 돌려준다 — `return <이름> ..` 그 이름은 0 에서 시작하는 지역이므로 몸이 한 번도 쓰지 않아도 돌려줄 값이 있다. 다른 값을 돌려주어도 된다(이름은 지역일 뿐이다). 모든 길이 `return` 으로 끝나야 하는 것은 이름 없는 출력과 같다(문단 3).

2 값을 돌려주지 않는 op(`output void`)은 `return` 만 적어 일찍 끝낼 수 있다.

2a 값을 돌려주지 않는 op 은 `return` 없이 **몸의 끝까지 진행해도** 된다. 돌려줄 값이 없으므로 끝나는 자리가 곧 돌아가는 자리다.

3 **값을 돌려주는 op** 은 모든 길이 값을 돌려주고 끝나야 한다. 값 없이 끝나는 길이 있으면 적합하지 아니하다.

다음 문장은 값을 돌려주고 끝나는 것으로 본다 — `return`, 두 갈래가 모두 값을 돌려주는 `if`…`else`, 그리고 **모든 갈래가 값을 돌려주는** `match`. `match` 가 모든 경우를 덮는지는 따로 검사된다(§6.6).

`guard`·`while`·`for` 는 빠져나가는 길이 있으므로 그 자체로는 값을 돌려주는 것으로 보지 아니한다.

몸이 기계 명령인 op(§6.9)은 값을 `out reg` 로 내므로 이 조항의 대상이 아니다.

> **쉬운 말로**
>
> 이 규칙이 없으면 값 없이 끝나는 길에서 처리기가 **0 을 대신 넣게 된다.** 그 0 은 소스 어디에도 없으므로, 프로그램을 다 읽고도 결과를 알 수 없다(§1.3). 그래서 이 언어는 그런 길을 아예 번역하지 않는다.

> **예제 (example) — 값을 안 내는 op 은 `return` 없이 끝나도 된다**
>
> **소스 코드 (source)**
>
> ```
> module ex_void .
>
> proc hold input n u8 . output void . effects none .
> do
>   let x u8 n. .
> end .
> ```

> **쉬운 말로**
>
> 돌려줄 값이 없으므로 **끝나는 자리가 곧 돌아가는 자리**다. 값을 돌려주는 op 이었다면 같은 모양이 거부된다(§6.5.5 문단 3).

6.5.6 블록과 들여쓰기

블록은 `do` 로 열고 `end` 로 닫는다. 들여쓰기는 **뜻이 없다** — 읽는 사람을 위한 것이다.

블록 안에서 지은 이름은 그 블록 안에서만 보인다. 블록이 닫힌 뒤에 그 이름을 읽는 것은 적합하지 아니하다(`E-NAME-SCOPE`) — 들어가지 아니한 길에서는 그 이름에 값이 놓인 적이 없으며, 처리기가 0 을 대신 놓지 아니한다(§6.5.5 (3)).

> **거부되는 예제 (rejected) — 블록 안에서 지은 이름을 블록 밖에서 읽는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_name_scope .
>
> fn pick input a u64 . output u64 .
> do
>   if gt a. 1 . do
>     let big u64 mul a. 2 . .
>   end .
>   return big. .        rem 들어가지 아니한 길에는 `big` 이 없다
> end .
> ```
>
> 진단: `E-NAME-SCOPE`

> **쉬운 말로**
>
> 들여쓰기로 블록을 나누는 언어도 있다. 이 언어가 그러지 않는 이유는, 눈에 안 보이는 글자(공백과 탭)가 프로그램의 뜻을 바꾸면 **보이는 것과 뜻이 갈릴** 수 있기 때문이다. `do` 와 `end` 는 눈에 보인다.

6.5.7 실패를 말하는 세 가지 길

실패를 말하는 길은 셋뿐이며, 쓰임이 서로 다르다.

표 39 — 세 실패 채널

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` 효과	계약이 깨졌다	다룰 수 없다. 프로그램이 멈춘다

2 `result` 의 오류 타입에 들어가는 변형은 그 op 의 `errors` 절이 적은 것과 **같아야** 한다. 적지 않은 오류를 돌려주거나, 적어 놓고 안 돌려주는 것은 적합하지 아니하다.
3 `panic` 은 되돌아 풀리지 아니한다. 멈추는 자리에서 프로그램이 끝난다 — 중간에 잡아 이어 가는 길은 없다.

> **예제 (example) — 세 채널이 각기 다른 모양으로 답한다**

소스 코드 (source)

```
module ex_channels .

enum io_error do
  too_big .
end .

rem ① 고칠 수 있는 실패 — result.
fn halve input a u8 . output result u8 io_error .
  errors too_big gt a. 200 . .
do
  guard le a. 200 . else return error too_big . . .
  return ok div a. 2 . . .
end .

rem ② 값이 없음 — option.
fn lookup input k u8 . output option u8 .
do
  guard lt k. 3 . else return none . .
  return some mul k. 10 . . .
end .

rem ③ 계약이 깨짐 — 부르는 쪽이 약속을 어기면 멈춘다.
fn strict input a u8 . output u8 .
  requires le a. 200 . .
do
  return add a. 1 . .
end .
```

실행 결과 (output)

```
halve(100) = ok 50 · halve(250) = err too_big · lookup(2) = some 20 · lookup(7) = none ·
strict(10) = 11 · strict(250) → 트랩
```

> **쉬운 말로**

셋을 가르는 물음은 "부르는 쪽이 무엇을 할 수 있는가"다. 파일이 없는 것은 다른 파일을 열어 볼 수 있으니 `result` 다. 찾는 것이 목록에 없는 것은 그냥 없는 것이니 `option` 이다. 계약이 깨진 것은 부르는 쪽이 **이미 약속을 어긴 것**이라 고칠 수 있는 일이 아니다 — 그래서 멈춘다.

6.5.8 실패를 다루기 — `try` 와 소비형

1 `result` 를 돌려주는 op 을 부를 때, `try` 는 성공하면 값을 꺼내고 실패하면 **그 오류를 그대로 위로 넘긴다**.

2 `try` 를 쓰는 op 은 자기도 그 오류를 돌려줄 수 있어야 한다. 아니면 거부된다 (`E-TRY-NORESULT`) — 실패가 갈 곳이 없기 때문이다. 꼬리를 붙여 채널을 바꾸거나((4)), (3) 의 방법으로 **여기서** 다루면 그 op 은 `result` 를 돌려주지 않아도 된다.

3 `result` 와 `option` 을 소비하는 방법은 다음으로 닫혀 있다 — `try`(꼬리를 붙인 것도 포함한다, (4)) · 묻는 것(`is_ok` `is_error` `is_some` `is_none`) · 꺼내는 것 (`ok_value` `error_value` `some_value`) · 대신 쓸 값을 주는 것(`value_or`).

4 `try` 뒤에 꼬리를 붙여 **채널을 바꿀** 수 있다. 값은 그대로이고 담는 그릇만 바뀐다.

표 40 — 채널 전환

모양	무엇에서 무엇으로	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** — 왜 없는지를 말한다

5 채널 전환은 **타입을 바꾼다.** `try <식> else_none` 을 적은 자리의 타입은 `option` 이지 그 안의 값이 아니다. 값 타입을 적어 놓고 채널 전환을 쓰면 적합하지 아니하다.

> **참고 (informative)**
>
> 이것이 규범인 까닭은 한때 그렇지 아니하였기 때문이다. 처리기가 꼬리 낱말을 못 보아 `output u64` 라 적은 op 이 `some 7 .` **을 돌려주었고**, 그러고도 `--check` 는 초록이었다. 채널이 바뀌는 자리를 규범이 말하지 아니하면, 도구가 그 자리를 잊어도 아무도 모른다.

6 꺼내는 것은 **부분 연산**이다(§6.2.8). 없는 쪽을 꺼내면 트랩한다.

7 `value_or` 의 기본값은 **값이 없을 때만 평가된다**(지연평가). 그러므로 기본값 자리에는 비싼 계산이나 실패할 수 있는 계산을 적어도 된다 — 값이 있으면 그것은 돌지 아니한다.

> **참고 (informative)**
>
> `value_or some 7 . div 1 0 . .` 은 `7` 이다. 값이 있으므로 기본값 쪽은 돌지 않는다. 값이 없을 때에만 그 자리가 평가되며, 그때는 트랩이 실제로 일어난다. ☞ 2026-08-29 까지는 그렇지 아니하였다 — 도구가 기본값을 **먼저** 평가하여, 값이 있는 프로그램이 없어도 될 실패로 끝났다. 정본(§6.2.8 · DECISION-0003 C1)이 지연평가로 정해 둔 자리였고, 도구를 정본에 맞추어 고쳤다. 값은 달라지지 아니하며 달라지는 것은 **어떤 트랩과 효과가 일어날 수 있는가** 이다.

> **쉬운 말로**
>
> 실패를 확인하는 코드를 매번 손으로 쓰면 길어지고, 길어지면 빼먹는다. `try` 는 그 되풀이를 한 낱말로 줄이되 **실패가 위로 간다는 사실은 소스에 남긴다** — 조용히 무시되는 실패가 없다.

6.5.9 멈추기 — `panic`

1 `panic` 은 프로그램을 즉시 멈춘다(§5.4). 이것은 효과이므로, 쓰는 op 은 자기 계약에 그 효과를 적어야 한다.

2 순수한 op(`fn`)은 `panic` 을 쓸 수 없다. 다만 계약이 깨졌을 때 처리기가 일으키는 트랩은 이와 별개이며, 그것은 op 이 한 일이 아니다.

6.5.10 나누어 도는 되풀이 — `parallel`

1 되풀이에 `parallel <조각> split .` 절을 붙이면, 그 되풀이를 **조각을 나누어** 여럿이 함께 돌 수 있다고 밝히는 것이다.

2 처리기는 그 말을 믿지 아니하고 **확인한다.** 확인하는 것은 되풀이의 각 걸음이 서로에게 기대지 않는다는 것이며, 조건은 셋이다.

표 41 — 나누어 돌 수 있는 되풀이의 조건

조건	어기면	무엇이 어긋났는가
자기 몫만 읽는다	`E-PAR-READ`	남의 자리를 읽으면 그 값이 아직 옛것인지 새것인지가 **누가 먼저 도느냐**에 달린다
자기 몫만 쓴다	`E-PAR-WRITE`	두 걸음이 같은 자리에 쓰면 남는 값이 차례에 달린다
걸음을 넘어 사는 자리에 쓰지 아니한다	`E-PAR-CARRY`	그런 자리는 걸음들을 **묶는다**. 모아야 한다면 `reduce <쌓는 자리> <연산> .` 로 밝힌다

3 `parallel` 이 조각을 이름 부르는데 그 조각의 길이로 도는 되풀이가 없으면 거부된다 (`E-PAR-NOLOOP`) — 나눌 것이 무엇인지 알 수 없기 때문이다.

4 모으는 연산은 **묶음의 차례를 바꿔도 답이 같아야** 한다. 그렇지 않은 연산은 거부된다 (`E-PAR-ASSOC`). 부동소수점 수의 덧셈은 그렇지 않으므로 따로 막는다(`E-PAR-FLOAT`).

4a 쌓는 자리의 **시작값은 그 연산의 항등원**이어야 한다. 아니면 거부된다(`E-PAR-IDENTITY`). 나누어 돌면 조각마다 그 시작값에서 다시 시작하므로, 항등원이 아닌 값은 **조각 수만큼** 셈에 들어간다 — 곧 나눈 답이 하나씩 돈 답과 달라진다(위 (2) 의 약속이 깨진다). 항등원은 `add`·`bit_or`·`bit_xor` 가 0, `mul` 이 1, 부호 없는 폭의 `max` 가 0 이다.

> **쉬운 말로**
>
> 나누어 도는 것을 **밝히는 말**로만 두면, 그 말이 틀렸을 때 프로그램은 대개 잘 돌다가 어느 날 다르게 돈다. 그래서 이 언어는 밝힌 말을 확인한다 — 확인할 수 없으면 나누지 않는 것이 아니라 **번역을 거절한다.** 「빠르지만 가끔 틀림」 은 이 언어가 파는 물건이 아니다.

> **주의 —**
>
> 나누어 돌지 **않아도 답은 같다.** 위 조건은 나눌 수 있음의 조건이지 뜻을 바꾸는 절이 아니다. 그러므로 나누어 돈 답과 하나씩 돈 답은 언제나 같다 — `E-PAR-FLOAT` 이 있는 까닭이 바로 그것이다.

6.5.11 몸이 갖추어야 하는 것

1 값을 내놓겠다고 적은 op 은 **모든 길에서** 값을 내놓아야 한다(`E-RETURN-PARTIAL`). 어느 한 길이 값 없이 끝에 닿으면 적합하지 아니하다.

2 선언의 머리(`fn`·`proc`·`on`·`test` …)에는 `do … end` 몸이 있어야 한다 (`E-STMT-NODO`).

3 맨 바깥 자리에 오는 것은 **선언의 머리**로 시작해야 한다(`E-TOPLEVEL`).

4 폼 안에 올 수 없는 것이 오면 거부된다(`E-FORM-UNEXPECTED`). 머리 자리에 op 이 아닌 이름이 오는 것도 그렇다(`E-HEAD-NOT-AN-OP`).

> **쉬운 말로**
>
> (1) 이 없으면 「값을 내놓는다」 는 선언이 **어떤 길에서만** 참인 말이 된다. 부르는 쪽은 그 말을 믿고 값을 쓰는데, 그 길로 가면 값이 없다. 선언은 **모든 길에 대한 약속**이므로 한 길이라도 어기면 그것은 약속이 아니다.

거부되는 예제 (rejected) — 어떤 길에서 값이 없다

소스 코드 (source)

```
module ex_partial .

fn f input a u8 . output u8 .
do
  if gt a. 5 . do return 1 . end .
end .                      rem `a` 가 5 이하인 길에는 값이 없다
```

진단: E-RETURN-PARTIAL

6.5.12 오류는 적은 것만 난다

1 op 이 내는 오류는 errors 절이 적은 것 안에 있어야 한다(E-ERR-UNDECLARED).

2 errors 절이 그 op 의 오류 타입에 **없는 이름**을 적는 것도 거부된다 (E-ERR-UNDEF).

3 다른 op 의 실패를 넘겨 올리면 — try <op> 로, 또는 return <op> … 로 그 result 를 그대로 돌려주면 — 그 op 의 실패가 이 op 의 실패가 된다. 그래서 부른 op 이 **낼 수 있는 오류**가 이 op 의 errors 절(절이 없으면 이 op 의 오류 타입) 안에 있어야 한다 (E-ERR-UNDECLARED).

3a 부른 op 이 낼 수 있는 오류는 그 op 의 errors 절이 적은 것이다. 절이 없으면 그 op 의 **오류 타입 전체**다 — 절을 적지 않는 것은 «이 타입의 오류는 무엇이든 날 수 있다» 는 뜻이고, 부르는 쪽도 그 뜻 그대로 읽는다.

주의 —

두 규칙은 같은 문장의 양쪽이다 — **적은 것과 내는 것이 같아야 한다.** 한쪽이 넘치면 부르는 쪽이 못 본 실패가 오고, 다른 쪽이 넘치면 오지 않을 실패를 다루게 된다 (§6.4.12 (4) 와 같은 까닭이다).

거부되는 예제 (rejected) — 적지 않은 오류를 낸다

소스 코드 (source)

```
module ex_err_undeclared .

enum e do
  bad .
end .

fn f output result u8 e .
do
  return error bad . .   rem `errors bad …` 를 적지 않았다
end .
```

진단: E-ERR-UNDECLARED

거부되는 예제 (rejected) — 남의 실패를 return 으로 넘긴다

소스 코드 (source)

```
module ex_err_handed_on .

enum parse_error do
  bad_digit .
end .

enum load_error do
  too_long .
end .
```

```
fn read_digit input c u8 . output result u8 parse_error .
errors bad_digit .
do
  guard le c. 9 . else return error bad_digit . . .
  return ok c. . .
end .

fn load_byte input c u8 . output result u8 load_error .
errors too_long .
do
  guard le c. 200 . else return error too_long . . .
  return read_digit c. . .     rem `bad_digit` 은 `load_byte` 의 약속에 없다
end .
```

진단: `E-ERR-UNDECLARED`

거부되는 예제 (rejected) — 절 없는 op 이 넘겨받은 실패를 다시 넘긴다

소스 코드 (source)

```
module ex_err_clauseless .

enum parse_error do
  bad_digit .
end .

enum load_error do
  too_long .
end .

fn read_digit input c u8 . output result u8 parse_error .
errors bad_digit .
do
  guard le c. 9 . else return error bad_digit . . .
  return ok c. . .
end .

fn digit_or_fail input c u8 . output result u8 parse_error .
do
  return read_digit c. . .     rem 절이 없다 — `parse_error` 의 무엇이든 날 수 있다
end .

fn load_byte input c u8 . output result u8 load_error .
errors too_long .
do
  guard le c. 200 . else return error too_long . . .
  let v u8 try digit_or_fail c. . . .     rem `parse_error` 는 `load_byte` 의 약속에 없다
  return ok v. . .
end .
```

진단: `E-ERR-UNDECLARED`

6.6 갈래를 가르기 — `match`

`match` 는 `enum` 값이 어느 갈래인지 보고 그에 맞는 블록을 실행한다.

`match` 는 **모든 갈래를 덮어야 한다.** 하나라도 빠지면 번역이 거부된다.

`match` 는 `enum` 뿐 아니라 정수도 가를 수 있다. 그때 한 갈래는 값 하나이거나 **범위** 이며, 범위는 `<처음> to <끝>` 으로 적고 양 끝을 포함한다.

4 정수를 가를 때도 **모든 값이 덮여야** 한다. 타입이 가질 수 있는 값을 범위들이 빈틈없이 덮거나, 나머지를 받는 갈래(_)가 있어야 한다.

5 한 갈래가 값을 여럿 받으려면 `or` 로 잇는다.

6 앞선 갈래가 이미 덮은 것을 뒤의 갈래가 다시 적는 것은 거부된다 (`E-MATCH-REDUNDANT`). 그 갈래는 **결코 돌지 않으며**, 돌지 않는 코드가 거기 있는 것은 쓴 사람의 뜻과 프로그램의 뜻이 갈렸다는 표시다.

7 `case` 에 적은 **맨 이름은 언제나 갈래**다. 선언된 갈래가 아니면 거부된다 (`E-MATCH-UNDEF`). 무엇이든 잡으려면 `_` 를 쓴다.

> **쉬운 말로**
>
> 2026-09-03 이전에는 맨 이름이 갈래가 아니면 **그 값을 묶는 이름**으로 읽혔고, 그래서 갈래 이름의 오타가 잡히기는커녕 **모든 것을 잡는 갈래**가 되어 (2) 의 빠짐 검사까지 만족시켰다. 갈래를 하나 더 늘려도 아무 말이 없었다 — 그 오타 갈래가 새 갈래까지 삼키기 때문이다. **처리기가 가장 쉽게 잡을 수 있는 잘못을 잡지 못하는 자리는 규칙이 아니라 구멍이다.**

7a 이름 뒤에 이름이 더 오면 그것은 **해체**다 — 값을 지닌 갈래를 푸는 것 (§6.2.19)이거나 짜임의 칸을 묶는 것이다. 그 자리의 첫 이름은 갈래가 아니어도 된다(짜임의 이름일 수 있다).

7b `case error <이름>` 의 이름이 그 오류 열거의 **선언된 갈래**면 그 갈래만 받는다. 선언된 갈래가 아니면 오류 값 전체를 그 이름에 묶는다(그 가지가 모든 오류를 받는다). (7) 이 `match` 전체에 대해 정한 규율 — 맨 이름이 갈래면 갈래다 — 을 이 자리에도 그대로 적용한 것이다.

> **쉬운 말로**
>
> 2026-09-16 이전에는 이 자리의 이름을 **언제나 묶음**으로 읽었다. 그래서 `case error not_digit .` 이 `empty` 오류까지 받았고, 갈래 둘 중 하나만 적은 `match` 가 빠짐 검사를 통과했으며, 갈래마다 적으면 둘째가 「이미 다뤘다」로 거부됐다(결함 노트 52). 갈래 이름을 적었는데 **다른 갈래가 그 가지로 들어오는 것**은 (7) 이 막으려던 바로 그 잘못이다.

8 값을 지닌 갈래(§6.2.19)를 가를 때, 묶는 이름의 수는 그 갈래가 지닌 것의 수와 **같아야** 한다(`E-MATCH-ARITY`).

8a `match` 의 마지막 자리에 `else do … end` 를 두면 그것이 **나머지를 받는 갈래**다 — `case _ .` 와 같은 일을 한다(부록 A.7 의 문법이 그 자리를 이미 적고 있다). 빠짐 검사는 그것을 덮은 것으로 세며, 그 뒤에 갈래를 더 적으면 (6) 에 걸린다.

8b 갈래의 패턴이 **중첩**일 때(`case ok (some x) · case ok none`), 그 바깥 꼬리표는 **안쪽 패턴이 스스로 빠짐없을 때** 덮인 것으로 센다. 안쪽이 빠짐없다는 것은 안쪽에 `_` 나 묶는 이름이 있거나, 안쪽이 `some` 과 `none` 을 다 적었거나, 안쪽 갈래들이 그 열거의 갈래를 다 적었다는 뜻이다.

> **쉬운 말로**
>
> (8b) 이전에는 중첩 갈래를 아예 **세지 않아서**, 빠짐없이 가른 `match` 가 `_` 를 요구받았다. 쓸모없는 `_` 는 나중에 갈래가 늘어도 아무 말을 하지 않으므로, 그것을 강요하는 것은 (2) 의 빠짐 검사를 **끄게 만드는 일**이었다.

9 `or` 로 이은 갈래들이 값을 묶으려면, **모든 갈래가 같은 자리에 같은 이름**을 지녀야 한다(`E-MATCH-ORBIND`). 그렇지 않으면 그 이름이 어느 갈래에서 왔느냐에 따라 다른 것을 가리키게 된다.

> **쉬운 말로**
>
> (2)·(4) 가 **빠짐**을 막는다면 (6) 은 **겹침**을 막는다. 둘은 같은 규율의 앞뒤다 — 갈래를 가르는 일은 값의 공간을 **빠짐없이, 겹침없이** 나누는 일이며, 어느 한쪽만 지키면 나눈 것이 아니라 그저 늘어놓은 것이다.

예제 (example) — 범위로 가른다

소스 코드 (source)

```
module ex_range_match .

fn band input a u8 . output u8 .
do
  match a. do
    case 0 to 9 do return 1 . end .
    case 10 to 255 do return 2 . end .
  end .
end .
```

실행 결과 (output)

```
band(5) = 1 · band(200) = 2
```

거부되는 예제 (rejected) — 덮이지 않은 값이 있으면 거부된다

소스 코드 (source)

```
module ex_range_gap .

fn band input a u8 . output u8 .
do
  match a. do
    case 0 to 9 do return 1 . end .
  end .
end .
```

진단: `E-MATCH-INEXHAUSTIVE`

쉬운 말로

빈틈을 남기면 번역이 거부되므로, "나머지는 무엇인가"를 저자가 반드시 말하게 된다. 다른 언어에서 이 자리는 조용히 아무 일도 안 하거나 예외가 되는데, 둘 다 소스를 읽고는 알 수 없다.

예제 (example) — match 는 모든 갈래를 덮는다

소스 코드 (source)

```
module ex_match .

enum color do
  red .
  green .
end .

export fn code input c color . output u32 .
do
  match c. do
    case red do return 1 . end .
    case green do return 2 . end .
  end .
end .
```

거부되는 예제 (rejected) — 갈래가 빠진 match

소스 코드 (source)

```
module ex_match_bad .

enum color do
  red .
  green .
end .

export fn code input c color . output u32 .
do
  match c. do
    case red do return 1 . end .
    rem `green` 을 안 다뤘다
  end .
end .
```

진단: `E-MATCH-INEXHAUSTIVE`

쉬운 말로

조건문을 이어 쓰는 것으로도 같은 일을 할 수 있다. `match` 가 나은 점은 **완전성** 하나다 — 갈래를 하나 더 만들었을 때, 그것을 안 다룬 자리를 처리기가 전부 찾아 준다. 조건문 사슬은 그 자리를 찾아 주지 않는다.

참고 (informative)

새 갈래를 더하는 일은 흔하다. 그때 "이 갈래를 어디어디에서 다뤄야 하지"를 사람이 기억해야 한다면 반드시 빠뜨린다. `match` 는 그 기억을 처리기에게 넘긴다.

6.7 시험 — `test` 와 `expect`

`test` 블록은 그 모듈의 시험을 적는 자리다. 처리기는 이것을 돌릴 수 있다.
`expect` 는 시험의 단언이다. 조건이 거짓이면 그 시험이 실패한다.
처리기는 `expect` 를 **최적화로 없애지 아니한다.** 사라진 시험은 돌지 않은 시험이다.
번역만 하는 것은 시험을 돌리지 아니한다. 번역이 통과했다는 것은 "시험이 통과했다"가 아니다.

예제 (example) — 시험

소스 코드 (source)

```
module ex_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 . .
end .
```

> **주의 — 계약 위반과 시험 실패는 다르다**
>
> 계약 위반은 **코드가 자기 약속을 어긴 것**이고, 시험 실패는 **시험이 코드가 틀렸다고 말하는 것**이다. 둘은 다른 진단으로 나온다 — 무엇을 고쳐야 하는지가 다르기 때문이다.

6.8 번역 시점의 매개변수 — `comptime`

1 `comptime` 을 붙인 매개변수는 **번역 시점에 값이 정해져야** 한다. 타입도 그렇게 받을 수 있다.

2 같은 op 을 여러 타입에 쓰려면 이 방법을 쓴다. 처리기는 쓰인 조합마다 실물을 만든다.

3 그 만들어짐은 **소스에 보인다.** 꺾쇠도 없고 추론도 없다 — 어떤 조합이 만들어졌는지가 부르는 자리에 적혀 있다.

4 `comptime` 자리에 번역 시점에 알 수 없는 값을 주면 적합하지 아니하다.

4-1 번역 시점에 아는 값은 정수 리터럴, 모듈 상수, `comptime <식>`(§6.8.1 의 셈), 그리고 같은 op 안의 지역 `let` 으로 그 값이 §6.8.1 (1a) 처럼 셈해지는 것이다. `var` 와 실행 값은 아니다. 레인 수(`vec <t> <n>`)로 쓰이는 매개변수에는 정수 리터럴과 모듈 상수만 준다 — 그 자리마다 실물이 만들어지기 때문이다.

4a 타입을 받는 `comptime` 자리를 **비워 두는 것**도 적합하지 아니하다(`E-MONO-NOTYPE`). 인자의 타입에서 되짚지 아니한다 — (3) 이 말하듯 무엇이 만들어지는지는 부르는 자리에 적혀 있어야 한다.

5 쓰인 조합마다 실물이 만들어지므로, 조합의 수가 곧 만들어지는 코드의 양이다. 이 비용은 부르는 자리에서 셀 수 있다.

> **예제 (example) — 번역 시점 값을 받는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_comptime .
>
> fn twice input comptime n u8 . input a u8 . output u8 .
> do
>   return add a. n. . .
> end .
>
> fn call_it output u8 .
> do
>   return twice 3 4 . .
> end .
> ```
>
> **실행 결과 (output)**
>
> ```
> call_it() = 7
> ```

> **거부되는 예제 (rejected) — 번역 시점에 알 수 없는 값은 줄 수 없다**
>
> **소스 코드 (source)**
>
> ```
> module ex_comptime_rt .
>
> fn twice input comptime n u8 . input a u8 . output u8 .
> do
>   return add a. n. . .
> end .
>
> fn call_it input k u8 . output u8 .
> do
>   return twice k. 4 . .
> end .
> ```
>
> 진단: `E-COMPTIME-ARG`

> **참고 (informative)**
>
> 다른 언어의 제네릭이 하는 일과 겹치지만, 여기서는 **암묵적인 것이 없다**. 무엇이 만들어지는지 소스만 보고 알 수 있어야 하고, 그것이 이 언어가 추론을 안 쓰는 이유다.

6.8.1 번역 시점에 셀 수 있어야 한다

1 `comptime` 자리의 식은 **번역할 때 값이 나와야** 한다. 지금의 처리기가 그것을 셈하지 못하면 거부된다(`E-COMPTIME-NONCONST`).

1a 지금의 처리기가 셈하는 것은 정수 리터럴, 모듈 상수, 값이 번역 시점에 나온 지역 `let`, `machine.*`, `native_lanes <타입>`, `config <이름>`, 그리고 순수한 정수 빌트인 (`add` · `sub` · `mul` · `div` · `mod` · `neg` · 비교 · `min` · `max` · 비트 연산 · `shl` · `shr` · 논리)이다. 지역 `let` 의 값이 번역 시점에 나온다는 것은, 그 값이 리터럴 · 이런 이름 하나 · `native_lanes <타입>` · `comptime <식>` 이라는 뜻이다. `var` 와 입력(`comptime` 입력도 — §6.8 의 실물은 하나이고 값은 실행할 때 건네진다)은 셈하지 아니한다. 셈이 값의 범위를 넘으면 (부호 있는 64 비트 · 부호 없는 64 비트의 넘침, 0 으로 나눔, 부호 없는 값과 음수를 섞음) 그 식은 셈하지 아니한다 — 실행할 때 멈출 식을 번역 때 다른 수로 바꾸지 않는다.

1c 순수한 `fn` 을 부르는 것도 셈한다. 처리기는 번역할 때 그 `fn` 을 실제로 돌리고, 돌린 답을 그 자리에 둔다. 효과를 낼 수 있는 `proc` 은 돌리지 아니한다. 입력은 정수나 `bool` 하나, 수 · `bool` 의 슬라이스, 구조체, 구조체의 슬라이스다. 정수 인자는 (1a) 처럼 셈할 수 있어야 하고, 슬라이스 인자는 번역 시점에 바이트가 정해진 것이어야 한다 — 따옴표 문자열 · `text` 리터럴(원문 그대로) · 번역 때 읽은 파일(1g) · 원소가 모두 상수인 `lit array` · `lit slice` · 배열이나 슬라이스 모듈 상수(1d)(1e). 구조체 인자는 바이트 배치가 있는 구조체(§6.2.6 (9))여야 하고, 칸마다 (1a) 처럼 셈할 수 있는 값을 적은 `lit s do … end` (안의 구조체도 그렇게 적는다)이거나 구조체 답(1f) — 모듈 상수나 그 자리의 부름 — 이다. `fn` 은 그 구조체를 프로그램의 경계에서 받는 것과 같이 받는다. 128 비트 칸에는 수 리터럴 하나를 적는다. 구조체의 슬라이스 인자는 그런 리터럴의 `lit slice s …` 이거나 구조체의 배열 · 슬라이스 답(1d)(1e)이다 — 슬라이스 칸이 있는 구조체의 답이면 처리기가 그 칸이 가리키는 바이트를 함께 건넨다. `mut` 입력은 받지 아니한다. 그 `fn` 이 넘치거나 계약을 어기거나 정해진 걸음 안에 끝나지 않으면, 또는 인자를 번역 시점에 알 수 없으면 `E-COMPTIME-NONCONST` 로 거부되며, 메시지가 그 까닭을 말한다.

1c-1 정해진 걸음은 처리기가 밝히는 수이며, 빌드 설정이 바꾼다 — 매니페스트의 `build comptime_steps <n> .`, 그 위에 명령줄 `--comptime-steps <n>`. 번역 때 돈 답을 간직해 다음 번역에 다시 쓸 수 있으며, 간직한 답은 그 `fn` 의 정의 · 입력 바이트 · 처리기 판 · 대상 · 걸음 예산이 모두 같을 때만 쓴다. 간직이 있어도 없어도 답과 거부는 같다.

1d 답이 배열인 `fn`(§6.2.6 (1c))도 돌린다. 원소는 크기 있는 수 · `bool` · 그런 원소의 줄(배열 안의 배열) · 구조체(§6.2.6 (9) · (9-1))다. 처리기는 결과의 자리를 주고 그 `fn` 을 돌리며, 돌린 뒤의 바이트는 프로그램에 박힌 **읽기 전용** 바이트가 된다 — 원소가 모두 상수인 나열과 같은 자리다 (§6.2.6 (7) ⓑ). 그 답에는 `let x array n t comptime f … .` 처럼 `f` 의 결과와 길이 · 원소 타입이 같은 이름을 붙인다. 몸 안의 `let` 이어도, 모듈 상수여도 된다. 그 이름으로 원소를 쓸 수 없다 — `set` · `copy` · `swap` 은 거부되고(`E-TYPE-MUT` · 모듈 상수의 `set` 은 `E-IMMUTABLE`), `mut` 자리에 넘기면 거부된다(`E-TYPE-ARGMUT`). `var` 에 묶거나, 길이나 원소 타입이 다른 이름에 묶거나, 식의 한가운데에 두면 거부된다(`E-RESULT-PLACE`). 내용이 같은 두 답이 같은 자리인지는 ⓑ 와 같이 **자유 선택 (free choice)** 이다(FC-0003).

1d-1 번역 때 도는 것은 순수한 `fn` 뿐이므로 그 `fn` 은 새 저장소를 짓지 아니한다 — 답의 자리는 바깥(처리기)이 주고, 답 안의 슬라이스 칸은 `fn` 의 입력이나 프로그램의 상수 바이트를 가리킨다. 처리기는 가리키는 바이트를 (1d) 의 읽기 전용 자리에 두고 칸의 주소를 그 자리로 다시 적는다. 칸이 다른 곳(그 `fn` 의 틀 안 바이트)을 가리키면 거부된다. 답의 개수를 미리 알 수 없으면 넉넉한 길이를 적고, 채운 수는 따로 셈한다(아래 예).

1e 답이 수 · `bool` · 구조체의 슬라이스(`output slice t .`)인 `fn` 도 돌린다. 답의 바이트를 베껴 (1d) 와 같은 **읽기 전용** 바이트로 둔다 — 그 `fn` 이 문자열을 가리키는 답을 냈든 입력의 조각을 냈든 같다. 원소가 슬라이스 칸이 있는 구조체이면 (1d-1) 과 같이 칸의 주소를 다시 적는다. 이름은 `let x slice t`

`comptime f … .` 로 붙이며, 몸 안의 `let` 이어도 모듈 상수여도 되고, 쓰기는 (1d) 와 같이 거부된다. 줄의 슬라이스(`output slice (array m t) .`)는 돌리지 아니한다(E-COMPTIME-NONCONST) — 줄의 답은 배열(1d)로 낸다.

1f 답이 구조체인 `fn` 도 돌린다. 답은 그 구조체의 **값**이다 — 칸마다 돌린 값을 상수로 적은 `lit s do … end` 와 같다(창 `view s` 가 아니다). 칸은 수 · `bool` · 수의 배열과 슬라이스(그 바이트는 (1d) 의 읽기 전용 자리에 둔다) · 그런 칸만 가진 구조체이다. 128 비트 칸은 위 절반까지 그대로 든다. `option` · `result` · 참조를 품은 칸이 있으면 돌리지 아니한다(E-COMPTIME-NONCONST).

1g `include_text "<경로>"` 는 번역할 때 그 경로의 파일을 읽은 바이트(`slice u8`, 읽기 전용)이다. 경로는 이 글이 든 소스 파일의 폴더에서 보며, 그 폴더 안에 머문다 — 절대 경로와 `..` 은 거부되고, 링크를 따라가 그 폴더 밖에 닿아도 거부된다. 이 꼴은 번역 때 도는 `fn` 의 입력 자리(1c)에서만 선다. 다른 자리에 두면 거부된다(E-INCLUDE-PLACE) — 실행할 때 파일을 읽는 것처럼 읽히기 때문이다. 파일을 읽지 못하거나 폴더 밖을 가리키거나 처리기가 밝힌 크기를 넘으면 그 `comptime` 자리가 그 까닭을 말하며 거부된다. 그 바이트는 (1c-1) 의 입력 바이트이다 — 파일이 바뀌면 간직한 답을 쓰지 아니한다. 처리기가 잠금 파일로 의존을 고정하는 빌드에서는 읽은 파일도 의존처럼 **내용**으로 고정된다: 항목의 이름은 `include:<매니페스트 폴더에서 본 경로>` 이고, 내용이 다르면 E-DEP-HASH, 항목이 없으면 E-DEP-UNLOCKED 로 거부된다. 고정하는 항목은 `--lock-write` 가 적는다.

1b 셈한 답은 **상수**다. 받는 자리는 그 답을 리터럴처럼 잰다 — 자리의 타입에 들지 않으면 리터럴과 같이 거부된다(E-TYPE-WIDTH). 답에 쓰인 이름의 타입은 답의 타입이 되지 아니한다.

2 모듈 자리의 `let` 은 **이름 붙인 상수**다. 그 값도 번역 시점에 나와야 한다 (E-CONST-NOTCOMPTIME). 값은 정수 리터럴이거나 `comptime <식>` 이다. 그 식은 다른 모듈 상수를 읽을 수 있고, 읽는 상수가 앞에 있든 뒤에 있든 같다. 서로를 읽어 값이 나오지 않으면 거부된다.

> **주의 —**
>
> (1) 이 말하는 것은 「이 식이 상수가 아니다」가 아니라 「**오늘의 처리기가** 이것을 셈하지 못한다」이다. 둘은 다르며, 이 문서는 그 차이를 숨기지 아니한다 — 못 하는 것을 원래 안 되는 것처럼 적으면, 나중에 할 수 있게 되어도 아무도 모른다.

> **거부되는 예제 (rejected) — 번역할 때 값이 나오지 않는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_comptime .
>
> fn f input a u8 . output u8 .
> do
>   let b u8 comptime a. . .    rem `a` 는 실행할 때에야 정해진다
>   return b. .
> end .
> ```
>
> 진단: E-COMPTIME-NONCONST

> **예제 (example) — 상수를 상수로 셈한다**
>
> **소스 코드 (source)**
>
> ```
> module ex_comptime_names .
>
> let KIB u64 1024 .
> let PAGE u64 comptime mul 4 KIB. . . .    rem 다른 모듈 상수를 읽는다
>
> fn f output u64 .
> do
> ```

```
  let extra u64 4 .
  return comptime add PAGE. extra. . . .
end .
```

실행 결과 (output)

```
f() = 4100
```

예제 (example) — 순수한 함수를 번역할 때 돌린다

소스 코드 (source)

```
module ex_comptime_fn .

fn fact input n u64 . output u64 .
do
  var r u64 1 .
  var i u64 1 .
  while le i. n. . do
    set r. mul r. i. . .
    set i. add i. 1 . .
  end .
  return r. .
end .

fn f output u64 .
do
  return comptime fact 10 . . .    rem 실행할 때에는 수 하나다
end .
```

실행 결과 (output)

```
f() = 3628800
```

예제 (example) — 표를 번역할 때 짓는다

소스 코드 (source)

```
module ex_comptime_table .

fn crc_table output r array 256 u32 .
do
  var n u32 0 .
  while lt n. 256 . do
    var c u32 n. .
    var k u32 0 .
    while lt k. 8 . do
      if eq bit_and c. 1 . 1 . do set c. bit_xor 3988292384 shr c. 1 . . .
      end else do set c. shr c. 1 . . end .
      set k. add k. 1 . .
    end .
    set idx r. cast usize n. . . c. .
    set n. add n. 1 . .
  end .
  return r. .
end .

let CRC array 256 u32 comptime crc_table . . .    rem 실행할 때에는 읽기 전용 표 하나다
```

```
fn f output u32 .
do
  return idx CRC. 255 . .
end .
```

실행 결과 (output)

```
f() = 755167117
```

예제 (example) — 글을 번역할 때 다듬는다

소스 코드 (source)

```
module ex_comptime_slice .

fn trim input s slice u8 . output slice u8 .
do
  var a usize 0 .
  var b usize len s. . .
  while and lt a. b. . eq idx s. a. . 32 . . do set a. add a. 1 . . end .
  while and gt b. a. . eq idx s. sub b. 1 . . 32 . . do set b. sub b. 1 . . end .
  return subslice s. a. b. . .
end .

let NAME slice u8 comptime trim "  hi  " . . .    rem 읽기 전용 바이트 "hi"

fn f output usize .
do
  return len NAME. . .
end .
```

실행 결과 (output)

```
f() = 2
```

예제 (example) — 메시지 표를 번역할 때 짓는다 — 할당 없이

소스 코드 (source)

```
module ex_comptime_messages .

struct msg do
  ko slice u8 .
  en slice u8 .
end .

fn parse input txt slice u8 . output r array 4 msg .
do
  var row usize 0 .
  var start usize 0 .
  var comma usize 0 .
  var i usize 0 .
  while le i. len txt. . . do
    if or eq i. len txt. . . eq idx txt. i. . 10 . . do
      if gt i. start. . do
        set idx r. row. . lit msg do ko subslice txt. start. comma. . . en subslice txt. add
comma. 1 . i. . . end . .
```

```
        set row. add row. 1 . .
      end .
      set start. add i. 1 . .
    end else do
      if eq idx txt. i. . 44 . do set comma. i. . end .
    end .
    set i. add i. 1 . .
  end .
  return r. .
end .

let MSGS array 4 msg comptime parse "hi,hello\nthx,thanks" . . .    rem 칸은 입력 글의 조각을 가리킨다

fn f output usize .
do
  return len field idx MSGS. 1 . en . . .
end .
```

실행 결과 (output)

```
f() = 6
```

예제 (example) — 구조체를 건네고 구조체의 조각을 받는다

소스 코드 (source)

```
module ex_comptime_structs .

struct pt do
  x u16 .
  y u16 .
end .

fn scale input p pt . input k u16 . output pt .
do
  return lit pt do x mul field p. x . k. . . y mul field p. y . k. . . end . .
end .

fn from input ps slice pt . input k usize . output slice pt .
do
  return subslice ps. k. len ps. . . .    rem 입력의 조각 — 새 저장소를 짓지 않는다
end .

let P pt comptime scale lit pt do x 3 . y 4 . end . 10 . . .
let REST slice pt comptime from lit slice pt lit pt do x 1 . y 2 . end . lit pt do x 5 . y 6 . end . . 1 . . .

fn f output u16 .
do
  return add add field P. x . field P. y . . field idx REST. 0 . x . . .
end .
```

실행 결과 (output)

```
f() = 75
```

거부되는 예제 (rejected) — 번역 때 읽는 파일은 번역 때 도는 함수의 입력이다

소스 코드 (source)

```
module ex_include_place .

fn f output usize .
do
  return len include_text "words.txt" . . .    rem 실행할 때 읽는 것처럼 읽힌다
end .
```

진단: `E-INCLUDE-PLACE`

거부되는 예제 (rejected) — 셈한 답이 받는 자리에 들지 않는다

소스 코드 (source)

```
module ex_comptime_width .

let K u64 6 .

fn f output u8 .
do
  return comptime mul K. 100 . . .    rem 600 은 `u8` 에 들지 않는다
end .
```

진단: `E-TYPE-WIDTH`

6.9 검사할 수 없는 자리 — `unsafe` 와 `extern`

1 `unsafe` 는 언어가 검사할 수 없는 일을 하는 자리를 표시한다. 이것은 **효과**이므로 계약에 적어야 하며(§7.1), 부르는 쪽으로 번져 올라간다.

2 `extern` 은 다른 언어로 쓴 것과 만나는 자리를 표시한다(§7.4).

3 이 두 자리는 **좁아야** 한다. 넓히면 그만큼 언어의 보장이 닿지 않는 넓이가 는다.

쉬운 말로

「안전하지 않다」는 말이 "아무렇게나 해도 된다"는 뜻이 아니다. **여기서부터는 언어가 아니라 사람이 책임진다**는 표시이며, 그 표시가 소스에 남아 있어서 나중에 찾을 수 있다.

6.9.1 기계 명령을 직접 쓰는 자리 — `asm`

1 `op` 의 몸을 **기계 명령**으로 쓸 수 있다. `asm <기계> .` 절을 선언에 적고, 몸에는 그 기계의 어셈블리를 적는다.

2 처리기는 그 안을 **읽지 못한다.** 어셈블러가 읽는다. 곧 이 언어가 지키는 모든 불변이 그 안에서 깨질 수 있고, 처리기는 그것을 볼 수 없다. 볼 수 없는 것은 **가둔다** — 다음 넷을 **모두** 갖추어야 한다.

표 42 — `asm` 을 쓰는 `op` 이 갖추어야 하는 것

갖출 것	왜
`unsafe` 표시	못 보는 것을 정직하게 표시한다
`cap machine`	기계 명령을 낼 **권리를 건네받는다** — 주변에 떠도는 권위가 아니다
`effects unsafe`	시그니처에 실려 **부르는 쪽이 안다**
기계 이름	어셈블리는 이식되지 아니한다. 처리기는 이식되는 척하지 아니한다

3 기계 이름은 §5.6 의 닫힌 집합에서 온다. 그 밖의 이름은 거부되며(E-ASM-TARGET), 지금 짓고 있는 기계와 **다른** 기계의 어셈블리도 거부된다 — 처리기는 그 op 을 조용히 버리지도, 다른 기계의 명령을 어셈블러에 넘기지도 아니한다.

4 피연산자는 `<레지스터> <이름>` 이 입력이고 `out <레지스터> <이름>` 이 출력이다. 레지스터만 적고 실을 것을 적지 아니하면 거부된다(E-ASM-OPERAND).

5 `clobber` 절은 이 명령이 **망가뜨리는 것**을, `options` 절은 처리기에게 주는 약속을 적는다.

> **주의 —**
>
> `asm` 은 절 낱말이므로 값 자리에 올 수 없다. 그래서 권한의 이름은 `cap asm` 이 아니라 `cap machine` 이다 — 이름이 부딪히는 것은 발견해서 고치는 것이 아니라 **설계로 없앤다**.

6 몸은 **어셈블리 그 자체**다 — 텍스트 리터럴 하나이며 그 밖의 것은 올 수 없다 (E-ASM-BODY). 기계 명령과 낮춘 코드를 섞을 수는 없다.

7 처리기는 템플릿 안을 읽지 못하지만, **피연산자 목록과 템플릿의 `{이름}` 은 같은 것을 말하는 두 표현**이므로 그 둘이 맞는지는 **양쪽으로** 검사한다.

표 43 — `asm` 이 거부되는 자리

진단	무엇이 어긋났는가
E-ASM-UNBOUND	템플릿이 부르는 이름을 `asm` 절이 **선언한 적 없다.** 어셈블러는 그것을 거절하거나, 더 나쁘게는 **그 자리에 있던 레지스터를 그냥 읽는다**
E-ASM-UNUSED	선언한 피연산자를 템플릿이 **쓰지 않는다.** 레지스터를 태우고, 어셈블리가 하지 않는 말을 한다 — 검사되지 않는 중복은 거짓말로 썩는다(⟦§1.1⟧)
E-ASM-BIND	입력 피연산자가 값이 **아닌 것**(매개변수도 지역도 아닌 이름)을 가리킨다. 값이 아니면 그 자리에 있던 것이 실린다
E-ASM-BODY	몸이 텍스트 리터럴 하나가 아니다
E-ASM-OPERAND	레지스터만 적고 실을 것을 적지 아니하였다
E-ASM-NOUNSAFE · E-ASM-NOCAP · E-ASM-NOEFFECT	위 표의 갖출 것 셋 가운데 빠진 것이 있다
E-ASM-TARGET · E-ASM-TARGET-UNKNOWN	기계 이름이 지금 짓는 기계와 다르거나, ⟦§5.6⟧ 의 닫힌 집합 밖이다
E-ASM-OPTLIE	`options pure` 가 **효과 줄과 어긋난다**

8 `options pure` 는 "이 어셈블리에는 곁효과가 없다"는 말이며, 그 말을 믿고 처리기는 이 명령을 **지우거나·끌어올리거나·두 번 대신 한 번만** 돌릴 수 있다. 그런데 효과 줄이 입출력·상태·장치를 만진다고 말한다면 **둘 다 참일 수 없다.** 이때 처리기는 `options` 를 믿으므로, 어긋남은 거부된다(E-ASM-OPTLIE).

> **주의 —**
>
> 처리기가 **믿는** 말과 처리기가 **검사하는** 말이 다를 때, 위험한 쪽은 언제나 믿는 말이다. `options` 는 검사할 수 없는 약속이므로 그 약속이 다른 곳의 선언과 어긋나면 그 자리에서 막는다.

6.9.2 씨와 만나는 자리 — `extern` 의 경계

1 `extern` 이 붙은 op 은 **몸이 씨에 있다.** 그러므로 이쪽에 몸을 적으면 거부된다 (E-FFI-BODY) — 몸이 둘인 것은 프로그램이 아니라 아무도 답할 수 없는 물음이다.

1a 몸이 씨에 있는 `extern` op 은 **블록 선언**이다. `struct` 가 칸을 `do … end` 에 담듯, 절을 `do … end` 에 담는다: `extern proc <이름> do <절>* end .`. 절은 저마다 자기 점으로 닫는다 (§6.4.1 (3c)). 블록에는 절만 온다 — 문장이 오면 몸이 둘이다(`E-FFI-BODY`). `do` 없이 이름 뒤에 절을 늘어놓고 `end` 로 닫는 꼴은 거부된다(`E-STMT-NODO`) — `end` 는 자기 `do` 만 닫는다(§6.1.6 (2a)).

예제 (example) — 몸이 씨에 있는 op — 절을 do … end 에 담는다

소스 코드 (source)

```
module ex_extern_block .

unsafe extern proc c_abs do
  input k cap c .
  input a i64 .
  output i64 .
  effects unsafe .
  link "llabs" .
end .
```

거부되는 예제 (rejected) — do 없이 절을 늘어놓고 end 로 닫는다

소스 코드 (source)

```
module ex_extern_nodo .

unsafe extern proc c_abs input k cap c . input a i64 . output i64 .
  effects unsafe .
  link "llabs" .
end
```

진단: `E-STMT-NODO`

2 `export extern` 은 **반대 방향**이다. 씨가 우리를 부르므로 몸은 **이쪽 것**이며, 몸이 없으면 거부된다(`E-FFI-BODY`).

2a 씨에서 보이는 이름은 `lw` 뒤에 경로의 마디마다 `_<길이><마디>` 를 이어 붙인 것이다. 길이는 10 진수이고 여러 자리일 수 있다. 모듈 안의 op 은 [모듈, 이름], 타입에 딸린 op 은 [모듈, 타입, 이름], 모듈 선언이 없는 파일의 op 은 [이름] 이 경로다 — `module listlit` 의 `inside` 는 `lw_7listlit_6inside`, `module shapes` 의 `rect.area` 는 `lw_6shapes_4rect_4area` 다. 모듈 안의 이름은 종류와 상관없이 하나뿐이므로 (§6.10.3) 경로가 이름을 하나로 정하고, 길이를 적으므로 이름 속 밑줄이 마디의 경계로 읽히지 않는다. `lw_` 다음이 숫자인 이름은 이 규칙의 것이고, `lw_` 다음이 글자인 이름은 처리기 자신의 것이다.

2b 머리에 `link "<씨 이름>" .` 을 적으면 그 이름이 **그대로** 씨 이름이 된다. 그 이름은 씨가 받을 수 있어야 한다 — ASCII 글자·숫자·밑줄이고 숫자로 시작하지 않으며, 씨와 씨++ 의 예약어가 아니고, `lw_` 나 `__` 나 `_ +` 대문자로 시작하지 않아야 한다. 그렇지 않으면 거부된다(`E-LINK-NAME`). 한 번역 단위 안에서 두 export 가 같은 씨 이름이면 거부된다(`E-ABI-NAME-DUP`).

거부되는 예제 (rejected) — 씨 예약어를 씨 이름으로 적었다

소스 코드 (source)

```
module ex_link_keyword .

export fn inside input a u64 . output u64 .
  link "inline" .
do
  return add a. 1 . .
end .
```

> 진단: `E-LINK-NAME`

3 씨를 부르는 op 은 다음 셋을 **모두** 갖추어야 한다. 하나라도 없으면 거부된다.

표 44 — 씨를 부르는 op 이 갖추어야 하는 것

갖출 것	없으면	왜
`unsafe` 표시	`E-FFI-NOUNSAFE`	그 함수 안에서는 이 언어가 지키는 모든 것이 깨질 수 있고 처리기는 그것을 보지 못한다. 검사할 수 없는 것은 **적어도 표시되어야** 한다
`input k cap c .`	`E-FFI-NOCAP`	씨로 들어가는 것은 **건네받는 권리**다 — 무더기(⟦§8.4⟧)·장치·기계 명령(⟦§6.9.1⟧)과 같다. 주변에 떠도는 문은 문이 아니다
효과 줄(적어도 `unsafe`)	`E-FFI-NOEFFECT`	효과 줄은 **부르는 쪽이 무엇을 떠안는지 배우는 길**이다. 깨끗한 시그니처 뒤에 숨은 씨는 이 언어가 없애려는 바로 그 숨은 비용이다
`link "<심볼>" .`	`E-FFI-LINK`	씨 이름은 **다른 언어에 하는 약속**이며 약속은 저자의 것이다. op 이름에서 지어내면 op 을 고쳐 부르는 순간 다른 함수를 부르게 되고 그 사실이 소스 어디에도 없다

3a 권리는 **종류로** 따진다. `cap c` 가 아닌 다른 권리(할당기·파일 계통 …)를 받는 것으로는 이 자리가 열리지 아니한다(`E-FFI-CAPKIND`). 있음이 아니라 **무엇인가**가 권한을 만든다.

4 경계를 건너는 타입은 **씨의 ABI 가 표현할 수 있는 것**으로 한정된다. 그 밖의 것 (⟦option⟧ · ⟦result⟧ · 벡터 · 그릇)은 거부된다(`E-FFI-TYPE`). 씨에는 그런 것이 없고, 처리기는 **있는 척하지 아니한다.**

4a 저절로 사상되는 것은 **조각(slice)뿐**이다. `slice <타입>` 는 「`<타입>` 를 가리키는 포인터」와 「개수」 **두 인자**가 된다. 폭을 안다 — `slice u32` 는 `uint32_t*` 이지 바이트 포인터가 아니다. 그 밖은 모두 손으로 적는다.

4a-1 정수 인자는 **선언한 폭**으로 건넌다 — 8 · 16 · 32 비트 정수는 그 폭과 부호의 씨 정수 타입(`int8_t` · `uint8_t` · `int16_t` · `uint16_t` · `int32_t` · `uint32_t`)이고, 그 밖의 정수는 `long long` 이다. `extern` 의 결과도 선언한 폭이다. 처리기가 내는 헤더가 그 원형을 적으므로, 씨 쪽이 그 헤더를 포함하면 어긋난 원형은 번역 오류가 된다.

4b **권리는 건너가지 아니한다.** `cap` 은 값이 아니라 권한이므로 씨에게 넘길 것이 없다.

4c 바이트 배치가 있는 struct(§6.2.7 — 칸이 모두 크기 있는 수 · 배열 · 그런 struct)는 **값으로** 건너간다. 씨를 부를 때 넘기는 것도, 씨가 우리를 부를 때 받는 것도 돌려주는 것도 칸의 차례와 폭이 같은 씨의 struct 다. 큰 struct 를 숨은 포인터로 돌려주는지는 씨 컴파일러가 대상의 규약대로 정한다 — 이 언어는 그것을 서명에 드러내지 아니한다. 128 비트 정수 칸을 가진 struct 는 건너가지 아니한다(`E-INT128-PLACE`, §6.2.2 (3b-2)).

4d 씨가 부를 이름을 내는 op(`export`)의 입력이 ⟦option⟧ · 벡터 · 비트셋이거나 배치 없는 struct 이면, 또는 결과가 그런 것 · 조각이면 처리기는 씨 이름을 **내지 아니하고** 그렇다고 알린다(`W-EXPORT-NOSYM`) — 헤더도 그 이름을 약속하지 아니한다. 씨의 서명에 다른 타입으로 적어 두고 값을 버리는 일은 없다.

5 ABI 이름은 `c` **하나**다. `sysv·win64·aapcs64` 를 낱말로 두지 아니한다 — 그것들이 묻는 것은 "이 기계에서 씨가 무엇인가"이고, 그 답은 지을 기계(§5.6)가 이미 정한다. 낱말을 늘리면 같은 것을 말하는 길이 둘이 된다.

6 가변인자는 **한쪽 방향뿐**이다. 씨의 가변인자 함수는 부를 수 있으나, 씨가 우리 가변인자를 부를 수는 없다 — 인자의 수와 타입을 실행 중에 모르는 채로 하는 일이라 계약이 설 자리가 없다.

7 되부름(callback)에는 새 규약이 없다. `export extern` op 은 이미 씨가 부를 수 있으므로, 되부름은 **그 op 의 주소**(`unsafe_fn <op>`)를 값으로 넘기는 것이다. 문에 들어서는 자리에서 경계 계약을 검사한다.

7a 그러므로 `unsafe_fn` 이 가리킬 수 있는 것은 **`export extern` 인 op 뿐**이다. 그렇지 않은 op 을 가리키면 거부된다(E-FN-NOTEXPORT) — 씨가 부를 수 없는 문의 주소를 건네는 것은 주소가 아니라 물음이다.
7b 되부름으로 쓸 op 은 **권한을 요구할 수 없다**(E-FN-CAP). 되부름에 들어서는 것은 씨이고, **씨는 건넬 권한이 없다.** 권한이 필요한 일은 되부름 **밖**에서 하고, 되부름 안은 셈만 한다.
7c `unsafe_fn` 뒤에는 **op 의 이름**이 와야 하며(E-FN-VALUE), 그 이름의 op 이 실제로 있어야 한다(E-FN-UNDEF).

> **주의 —**
>
> 권한은 부르는 쪽이 건네는 것인데(§7.2), 되부름의 부르는 쪽은 씨다. 그러니 되부름이 권한을 요구하면 그 자리는 **아무도 채울 수 없는 자리**가 된다. 실행할 때 비어 있는 것을 발견하는 대신, 번역할 때 그 모양을 거절한다.

8 `owned <타입>` 를 `extern` 에 넘기면 **책임이 씨에게 옮겨간다.** 이쪽의 의무는 그 지점에서 끝나고, 그 뒤의 반납은 씨의 몫이다. 이것이 지켜졌는지는 **검증되지 아니한다** — 문 안은 우리가, 문 밖은 씨가 책임진다.
9 생 포인터(§8.5)가 경계에서 어긋나면 거부된다(E-UPTR-TYPE).
10 **내보내는 타입은 정렬을 기계에서 받아올 수 없다**(E-ABI-TARGET-ALIGN). `align machine.cache_line` 처럼 적으면 배치가 기계마다 달라지는데, 씨 쪽이 읽는 머리글은 배치가 **하나**라고 여긴다 — 둘은 아무 말 없이 갈라진다. 내보내는 타입에는 **적힌 수**로 정렬을 준다(`align N .`).

> **쉬운 말로**
>
> 이 절이 하는 일은 씨를 편하게 부르게 하는 것이 아니라, **씨를 부르는 자리를 좁고 보이게** 만드는 것이다. 표시(`unsafe`)·권리(`cap c`)·효과 줄 셋은 각각 사람에게, 처리기에게, 부르는 쪽에게 같은 사실을 말한다. 셋 중 하나만 있어도 「부를 수는 있다」 — 그러나 그때 이 언어는 자기가 무엇을 잃었는지 말하지 못하게 된다.

6.9.3 기계마다 다른 명령을 쓰는 자리 — `unsafe target`

1 어떤 op 은 **특정 명령집합이 있을 때만** 뜻이 있다. 그런 op 은 `unsafe target <명령집합>` 으로 그 사실을 적는다.
2 처리기가 모르는 명령집합 이름은 거부된다(E-TARGET-ISET). 이름은 닫힌 집합이다.
3 이름은 알되 **지금 짓는 기계가 그것을 주지 못하면** 거부된다(E-TARGET-INTRIN).
3a 이 절이 막는 것은 **그 op 이 서는 자리**이지 op 하나하나가 아니다. 어떤 잎이 이 울타리 안에 들어야 하는지는, 그 잎의 뜻이 **기계에 달려 있을 때**만 정해진다 — 어느 기계에서나 같은 값을 내도록 적힌 op 은 이 울타리에 들지 아니한다.

> **참고 (informative)**
>
> 이 자리에는 `avg` 를 «기계 붙박이» 로 묶어 두는 조항이 있었다. 지웠다(2026-09-17) — §6.3.7 이 그것을 **보통 레인 op** 으로 적고 있었고, 이 판의 두 뒤끝 모두 그것을 기계 명령이 아니라 넓힌 정수 산술로 내고 있었다. 곧 막고 있던 것은 **있지도 않은 기계 의존**이었다. 글과 도구가 갈린 자리에서는 글을 따른다.

> **쉬운 말로**
>
> 이 자리가 `unsafe` 인 까닭은, 그 op 의 뜻이 **기계에 달려 있기** 때문이다. 이 언어가 지키는 것들은 기계가 무엇이든 같아야 하는데, 여기서는 그 약속을 놓는다 — 그러니 놓는다는 사실이 소스에 적혀 있어야 한다.

> **주의 —**
>
> 호스트에 닿는 라이브러리 잎 가운데 오늘 **POSIX 로만** 내려가는 것들이 있다. 그런 잎을 그것이 없는 기계로 지으려 하면 거부된다(E-TARGET-LEAF) — 없는 것을 있는 척 내려보내지 아니한다.

6.9.4 정했으되 짓지 않은 것 — `raw` · `addr` · `adopt` · `clone`

1 이 조항의 넷은 **뜻이 정해져 있되 처리기가 아직 짓지 않은** 낱말이다. 처리기는 그 낱말을 알아보고, 쓰면 번역이 거부된다(`E-UNSAFE-UNBUILT`) — 받아서 다른 뜻으로 돌리거나 조용히 해석기로 떨어뜨리지 아니한다. `adopt` · `clone` · `addr` 는 예약어가 아니다: 저자가 그 이름의 op 을 지으면 그 op 이 선다.

2 `raw <T>` 는 **어느 저장소에 사는지 모르는 위치**다. `unsafe` 효과를 지닌 op 안에서만 만들고, 읽고 쓰고, 옮긴다(위치에 수를 더한다). 복제할 수 있다. `ref` · `mut_ref` · 조각에서 얻을 수 있고, 그때 그 참조가 지키던 배타(§8.12)와 수명의 보장을 버린다. 타입 자리에 `raw` 를 적으면 지금은 거부된다.

3 `addr` 는 **위치를 수로 본 것**이다 — 더하고 견줄 수만 있고, 그것으로 값을 읽거나 쓸 수 없다. `raw` 와 오가는 것은 `unsafe` 안에서만이다. `addr <지역>` 은 그 지역의 위치를 얻는다. 타입 자리에 `addr` 를 적거나 `addr <지역>` 을 쓰면 지금은 거부된다.

4 `adopt` 는 `raw <T>` 를 안전한 조각으로 되돌린다. 셋을 **모두** 요구한다: `unsafe` 안일 것 · 그 메모리가 사는 살아 있는 영역을 증인으로 받을 것(새 조각은 그 영역의 수명을 받는다) · 정렬과 범위를 계약(`requires`)으로 적을 것. 쓰기 조각으로 되돌리려면 «그 구간에 다른 접근이 없다» 는 가정을 `assume` 으로 적는다 — 처리기가 볼 수 없는 것을 사람이 진다는 표시다.

5 `clone <값> <영역>` 은 값을 그 영역에 **깊이 베낀다.** 숨은 복사가 없다는 원칙대로(§8) 깊은 복사는 이 낱말로만 일어난다. 할당 효과(`alloc`)를 지닌다.

6 **유예.** 넷은 쓰임이 나올 때 짓는다. 다른 길 — 바이트 위의 창(§6.2.13) · `extern`(§6.9.2) · 얼로케이터(§8.13) — 이 그 쓰임을 다 채워 넷이 **필요 없다고 보이면 버린다.** 짓을지 버릴지는 소유자가 정한다. 그때까지 이 조항은 남고 처리기는 그 낱말을 계속 알아본다.

> **거부되는 예제 (rejected) — 아직 짓지 않은 위치 타입**
>
> **소스 코드 (source)**
>
> ```
> module ex_unbuilt .
>
> fn f input p raw . output u64 . do
>   return 0 .
> end .
> ```
>
> 진단: `E-UNSAFE-UNBUILT`

> **참고 (informative)**
>
> 출처는 RFC-0005 §6.1 · §6.8(참조의 갈래와 위치 → 안전한 참조로 되돌리기)과 RFC-0049(`clone`)다. 두 RFC 는 채택된 뒤 보관됐는데, 이 넷은 정본으로 옮겨지지 않았다(2026-11-11 훑기). 소유자는 «정식으로 넣되, 차후 불필요하게 되면 버린다고 명시해서 계속 추적» 하기로 했다(2026-11-12).

6.9.5 정했으되 짓지 않은 특례 — 침투형 목록 · 그래프 · 한 명씩 주고받는 줄 · `layout c`

1 이 조항은 저수준 자료구조의 **특례**를 적는다. §6.9.4 와 같이 뜻은 정해져 있되 언어의 낱말로는 아직 짓지 않았다. 특례는 따로 선 세계가 아니라 **관리되는 저장의 저수준 모습**이다: 같은 배타 규칙(§8.12)과 같은 영역 규칙을 따르고, 특례만을 위한 빌림 규칙을 두지 아니한다.

2 **침투형 목록과 그래프** — 노드가 제 안에 이웃을 가리키는 칸을 지니는 목록 · 트리 · 그래프. 노드는 모두 **한 영역**에 살고, 이웃은 포인터가 아니라 그 영역의 핸들로 가리킨다. 그런 구조를 선언하는 타입은 만들기 · 넣기 · 빼기 · 돌기 · 풀기를 **반드시** 갖춘다 — 그 의무의 이름이 원안의 `T-LINKED` 이고, 그것을 적는 자리가 원안의 `manage` 절이다. 철자(`linked intrusive` · `graph intrusive` · `manage`)는 원안이 정하지 않고 남겼으므로 짓을 때 정한다. 그래서 낱말로 거부되지 않는다.

3 **한 명씩 주고받는 줄**(원안의 `queue spsc`) — 쓰는 흐름 하나와 읽는 흐름 하나만 닿는, 크기가 정해진 고리 줄. 잠금 없이 원자 연산으로만 주고받는다(§10). 철자는 (2) 와 같은 까닭으로 짓을 때 정한다.

4 `layout c` — 구조체를 C 의 규칙대로 놓는다. 그 일은 이미 기본 배치가 한다: 필드는 선언 차례대로, 제 정렬에 맞춰 놓이고 처리기는 차례를 바꾸지 아니한다(§6.2.15 (1)). 그래서 `layout c` 는 짓지 않았고, 배치의 낱말은 닫힌 둘(`packed` · `native`)이다 — `layout c .` 은 지금 거부된다(`E-TYPE-LAYOUT`).

5 **유예.** 이 특례들은 쓰임이 나올 때 짓는다. 지금 그 쓰임을 맡은 것: 침투형 목록 · 노드 저장은 라이브러리 `nodelist` · `pool`, 한 명씩 주고받는 줄은 라이브러리 `spsc`, C 의 배치는 기본 배치. 라이브러리로 적은 것이 처리기의 도움(타입 · 효과 · 하강) 없이는 안전하거나 빠르게 되지 않는다는 실측이 나오면 짓고, 라이브러리가 그 쓰임을 다 채운다고 보이면 **버린다.** 짓을지 버릴지는 소유자가 정한다.

> **거부되는 예제 (rejected) — `layout c` 는 짓지 않았다**
>
> **소스 코드 (source)**
>
> ```
> module ex_layout_c .
>
> struct header do
>   layout c .
>   magic u32 .
> end .
> ```
>
> 진단: `E-TYPE-LAYOUT`

> **참고 (informative)**
>
> 출처는 RFC-0050(저수준 특례 — D4 «침투형은 별도 패러다임이 아니다» · §6.6.2 `layout c`)과 RFC-0051(하드웨어 표면 정책), 그리고 RFC-0002 §6.4.1(`T-LINKED` · `manage`)이다. 셋은 채택된 뒤 보관됐는데, 이 특례들은 정본으로 옮겨지지 않았다(2026-11-11 훑기 F6). 소유자는 §6.9.4 와 같이 «정식으로 넣되, 차후 불필요하게 되면 버린다고 명시해서 계속 추적» 하기로 했다(2026-11-12).

6.10 모듈 (Modules)

1 소스 파일 하나가 모듈 (module) 하나다. 모듈은 자기 이름을 첫 줄에 적는다.

2 모듈 이름은 파일 이름과 같을 필요가 없다.

6.10.1 무엇을 내보내나

1 `export` 를 붙인 것만 다른 모듈이 쓸 수 있다. 붙이지 않은 것은 그 모듈 안에서만 쓴다.

2 곧 **감춘 것이 기본**이다. 내보내는 쪽이 그것을 적어야 한다.

2a `export` 는 선언에 붙는다: `fn` · `proc` · `struct` · `enum` · `type` · `newtype` · `actor` · `trait`. 모듈 상수(`let`) · 모듈 변수(`var`) · `test` 블록에는 붙지 아니한다(`E-TOPLEVEL`) — 상수는 모듈의 겉면이 아니고(op 으로 내준다), 시험은 그것을 가진 모듈의 것이다.

2b 내보낸 op 의 서명이 내보내지 않은 이름난 타입(구조체·열거·newtype·액터)을 쓰면 처리기는 그것을 알린다(`W-EXPORT-HIDDEN`) — 들여온 쪽은 그 타입을 적을 수 없으므로 값을 받을 이름을 지을 수 없고, 그 export 는 밖에서 쓸 수가 없다. 프로그램 자체는 적합하므로 거부가 아니라 알림이다. 투명 별칭(`type`)은 바탕 타입을 적으면 되므로 여기 들지 아니한다. **거부가 아닌 까닭**: 한 모듈만으로 이루어진 단위에서는 「밖」이 없어 그 export 를 쓸 수 없는 이가 아무도 없다. 이 단위가 라이브러리로 쓰일지 프로그램으로 끝날지는 처리기가 알 수 없으므로, 아직 해가 되지 않은 것을 거부하지 아니한다.

> **예제 (example) — 내보내기**
>
> **소스 코드 (source)**
>
> ```
> module ex_export .
>
> rem `export` 를 붙인 것만 다른 모듈이 쓴다.
> export fn visible input n u32 . output u32 .
> do
> ```

```
  return mul n. 2 . .
end .

rem 이것은 이 모듈 안에서만 쓴다.
fn hidden input n u32 . output u32 .
do
  return add n. 1 . .
end .
```

쉬운 말로

반대로 된 언어도 있다 — 기본이 공개이고 감추려면 표시하는 방식이다. 이 언어가 감춤을 기본으로 두는 이유는, **밖에서 보이는 것이 곧 약속**이기 때문이다. 실수로 약속하는 일보다 실수로 감추는 일이 고치기 쉽다.

6.10.2 무엇을 가져오나

1 `use` 는 다른 모듈을 가져온다. 가져올 것이 어디 있는지도 함께 적는다.
2 가져온 모듈의 이름을 통해 그 안의 것을 부른다.
3 가져올 자리에 그 모듈이 없으면 번역이 거부된다.

문법 틀 (grammar shape) — 가져오기 — 자리를 적는 갈래

```
module <내 모듈 이름> .

use <가져올 모듈 이름> from "<그 모듈의 자리>" .
```

4 자리를 적지 아니하는 갈래도 있다. 이때 그 모듈은 **같은 번역 단위 안에** 있어야 한다 — 여러 파일을 한 단위로 넘기면 그 안에서 찾는다.

문법 틀 (grammar shape) — 가져오기 — 같은 단위 안에서

```
module <내 모듈 이름> .

use <가져올 모듈 이름> .
```

4a `use <모듈> [from "<자리>"] as <별칭> .` 로 **그 모듈을 부를 이름을 바꿀 수 있다.** 별칭은 모듈 이름이 서는 모든 자리에 선다 — `op` 을 한정하는 자리(`<별칭>.<op>`)도, 타입을 적는 자리(`<별칭>.<액터>`)도 그렇다.
4b 별칭을 적으면 **원래 모듈 이름은 그 단위 안에서 서지 아니한다**(`E-USE-ALIASED`). 별칭은 덧이름이 아니라 **바꿔 부르기**다. 두 이름이 다 서면 같은 모듈을 두 철자로 부르는 코드가 생기고, 읽는 사람이 둘이 같은 것인지 확인해야 한다 — 별칭을 적은 까닭은 원래 이름이 불편해서인데, 그 이름이 계속 서 있으면 별칭이 한 일이 없다.

쉬운 말로

이 언어가 되풀이하는 규율 하나가 여기에도 선다: **한 뜻에 한 철자.** 같은 것을 적는 길이 둘이면 읽는 사람이 둘 다 알아야 하고, 고치는 사람은 둘 다 고쳐야 한다. `use` 줄 자신은 예외다 — 거기가 바로 원래 이름을 적는 자리이기 때문이다.

문법 틀 (grammar shape) — 가져오기 — 이름을 바꿔서

```
module <내 모듈 이름> .

use <가져올 모듈 이름> as <별칭> .
```

5 자리를 적지 아니한 이름이 그 단위 안에 없으면, 처리기는 **그 사실을 말하여야 한다.** 이 판의 처리기는 경고(W-USE-EXTERNAL)로 말한다 — 모듈을 찾아다니는 검색 경로가 없으므로, 그 이름이 있는지 없는지를 이 단위만 보고는 알 수 없기 때문이다. 그 이름을 실제로 쓰면 그때는 거부된다.

참고 (informative)

검색 경로가 없다는 것이 이 설계의 요점이다. 이름만 적어 두고 도구가 어딘가에서 찾아오게 하면, 그 프로그램이 무엇에 기대는지가 소스 밖의 지식이 된다 — 같은 소스가 기계에 따라 다른 것을 가져올 수 있다. 자리를 적거나, 같은 단위로 함께 넘기거나, 둘 중 하나다.

참고 (informative)

이 자리는 실제로 도는 예제 대신 **틀**로 적었다. 가져오기는 그 파일이 실제로 있어야 번역되므로, 어느 자리에서나 도는 예제를 만들 수 없다. 검사기가 컴파일해 보는 것은 #ex 뿐이며, 틀은 그 대상이 아니다.

참고 (informative)

어디서 가져오는지를 소스에 적는 것은 이 언어의 규율과 이어진다 — 이름만 적고 어디서 오는지는 도구가 알아서 찾는 방식이면, 그 프로그램이 무엇에 기대는지가 소스 밖의 지식이 된다.

6.10.3 이름이 부딪히면

1 한 모듈 안에서 같은 이름을 두 번 선언할 수 없다. 그렇게 하면 번역이 거부된다.

1a 모듈의 최상위 이름은 **한 이름 공간**에 든다 — 모듈 · fn·proc(extern 포함) · type · newtype · struct · enum 과 그 갈래 · actor · trait · 모듈 let/var · test. 종류가 달라도 같은 이름이면 거부된다(E-NAME-DUP). 칸 · 타입에 딸린 op(<타입>.<이름>) · actor 의 상태와 처리기 · trait 가 요구하는 op 은 그 그릇 안에 들며, 그릇이 다르면 같은 이름을 쓸 수 있다.

2 가져온 이름들이 서로 부딪히면, 부르는 자리에서 어느 것인지 밝혀 적어야 한다.

거부되는 예제 (rejected) — 같은 이름을 두 번 선언

소스 코드 (source)

```
module ex_dup .

fn f output u32 . do return 1 . end .
fn f output u32 . do return 2 . end .
```

진단: E-NAME-DUP

거부되는 예제 (rejected) — 종류가 달라도 한 이름 공간이다 — 모듈 let 과 fn

소스 코드 (source)

```
module ex_dup_kinds .

let limit u64 3 .
fn limit output u64 . do return 1 . end .
```

진단: E-NAME-DUP

6.10.4 선언의 차례

1 모듈 안의 최상위 선언은 **차례와 무관하다.** 뒤에 선언한 op 을 앞에서 부를 수 있고, 서로 부르는 op 도 쓸 수 있다.

2 op 본문 안에서는 그렇지 않다 — 지역 이름은 **쓰기 전에 선언해야** 한다.

> **쉬운 말로**
>
> 두 규칙이 다른 이유는 읽는 방식이 다르기 때문이다. 파일 전체는 훑어보며 읽지만, 함수 본문은 위에서 아래로 읽는다. 본문에서 아직 안 나온 이름이 쓰이면 읽는 사람이 되돌아가야 한다.

6.10.5 op 안의 op — 로컬 op

1 op 본문의 **바로 아래**(본문 `do … end` 의 문장 자리)에 `fn`·`proc` 선언을 둘 수 있다. 이것을 로컬 op (local op) 이라 한다. `if`·반복·블록 안이나 다른 로컬 op 안에 두는 것, 본문 안에 타입·actor· trait·`test`·모듈을 선언하는 것은 적합하지 아니하다(`E-LOCAL-PLACE`).

2 로컬 op 의 이름은 **그 op 의 본문 안에서만** 보인다. 그 안에서는 최상위 선언처럼 차례와 무관하다 — 선언 앞에서 부를 수 있고, 같은 op 의 로컬 op 끼리 서로 부르고 되돌아 부를 수 있다(§6.10.4 (1)). 다른 op 은 그 이름을 보지 못한다. 서로 다른 op 이 같은 로컬 이름을 쓸 수 있다.

3 로컬 op 은 바깥 op 의 매개변수·지역·region·빌림·능력을 **보지 않는다.** 그런 이름을 쓰는 것은 적합하지 아니하다(`E-LOCAL-CAPTURE`). 필요한 값은 매개변수로 건넨다. 로컬 op 은 보통 op 과 같은 머리(절)와 몸을 가지며, 효과·계약·오류·소유의 규칙도 같다. 로컬 `proc` 의 효과는 그것을 부르는 자리의 효과다(§7).

4 로컬 op 의 이름은 바깥 op 의 이름, 그 매개변수·지역, 모듈이 이미 가진 이름을 가리지 아니한다 (`E-NAME-SHADOW`). 한 op 의 로컬 op 끼리 이름이 같을 수 없고(`E-NAME-DUP`), 기본 연산의 이름을 쓸 수 없다(`E-NAME-BUILTIN`).

5 로컬 op 은 `export`·`extern`·`unsafe` 를 달 수 없다(`E-LOCAL-EXPORT`) — 밖에서 쓰려면 최상위에 선언한다.

> **예제 (example) — 쓰는 자리 가까이에 둔 보조 op**
>
> **소스 코드 (source)**
>
> ```
> module ex_local .
>
> export fn count_selected input xs slice u8 . input mask u8 . output u64 .
> do
>   fn has_bits input x u8 . input bits u8 . output bool .
>   do
>     return eq bit_and x. bits. . bits. . .
>   end .
>   return pipe xs. do
>     filter has_bits with mask. .
>     count .
>   end . .
> end .
> ```

> **거부되는 예제 (rejected) — 로컬 op 은 바깥의 이름을 보지 않는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_local_capture .
>
> export fn f input n u64 . output u64 .
> do
>   fn g input k u64 . output u64 .
> ```

```
  do
    return add k. n. . .
  end .
  return g 1 . .
end .
```

진단: E-LOCAL-CAPTURE

거부되는 예제 (rejected) — 로컬 op 은 본문 바로 아래에만 둔다

소스 코드 (source)

```
module ex_local_place .

export fn f input n u64 . output u64 .
do
  if gt n. 1 . do
    fn g input k u64 . output u64 .
    do
      return k. .
    end .
  end .
  return n. .
end .
```

진단: E-LOCAL-PLACE

참고 (informative)

로컬 op 은 클로저가 아니다. 이름이 보이는 범위만 좁을 뿐, 바깥에서 무엇도 몰래 가져오지 않으므로 처리기는 그것을 보통 op 처럼 다룬다 — 실행 중에 만들어지는 객체도, 숨은 인자도 없다.

6.11 타입에 붙은 op 과 트레이트 (Type-associated ops and traits)

6.11.1 타입에 op 을 붙이기

1 op 의 이름 앞에 타입 이름과 점을 붙이면, 그 op 은 **그 타입에 붙은 것**이 된다.
2 붙은 op 의 첫 입력은 그 타입의 값이다.

6.11.2 트레이트

1 트레이트 (trait) 는 어떤 타입이 갖춰야 할 op 의 **목록**이다. 각 op 의 이름과 시그니처를 적는다.

쉬운 말로

모양이 여럿이면(사각형 · 정사각형 …) 넓이를 구하는 **방법**은 타입마다 다르지만, "넓이를 알려 준다"는 **약속**은 같다. 트레이트는 그 약속에 이름을 붙인다. 그러면 "넓이를 알려 주는 것이면 무엇이든" 받는 op 을 **한 번만** 쓸 수 있다 — 타입마다 같은 op 을 베껴 쓰지 않아도 되고, 약속을 안 지킨 타입은 번역 때 걸린다. 부르는 자리는 그 타입이 무엇인지 몰라도 된다. 이 언어에는 상속이 없으므로, 여러 타입을 한 이름으로 다루는 길은 이것이다.

예제 (example) — 트레이트는 무엇에 쓰나 — 한 op 이 여러 타입을 받는다

소스 코드 (source)

```
module ex_trait_why .

rem 모양마다 넓이를 구하는 법은 다르다. 그러나 «넓이를 알려 준다» 는 약속은 같다.
```

```
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 .

rem 이 op 은 **어떤 모양이든** 받는다 — `requires shape t` 가 «넓이를 알려 주는 타입만» 이라고 못 박는다.
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 lit rect do w 2 . h 3 . end . .
  let q lit square do side 4 . end . .
  return add double_area rect. r. . double_area square. q. . . .
end .
```

실행 결과 (output)

```
demo() = 44
```

1a 트레이트는 op 을 **여럿** 적을 수 있다. 서명 하나는 op 의 **이름으로 시작**하고 그 op 의 절이 뒤따르며, 다음 이름이 다음 서명을 연다. 서명마다 한 줄에 적는 것이 관례다. 서명의 절은 op 머리와 **같은 차례**를 따른다(§6.4.1 (3a)) — 입력 · 출력 · 효과 차례다. 적는 op 의 수에는 한도가 없다.

1b 서명에는 `fn`·`proc` 을 적지 **아니한다.** 그 op 이 무엇을 할 수 있는지는 서명의 `effects` 줄이 정한다.

- `effects` 줄이 없으면 효과가 **없는** op 이다. 이것은 `fn` 으로도, 효과를 적은 `proc` 으로도 갖출 수 있다.

`proc` 은 언제나 `effects` 줄을 적는다(§7.1, 없으면 `E-EFFECT-MISSING`) — 효과 없는 서명을 갖추는 `proc` 은 `effects none .` 이다.

- `effects` 줄이 있으면 갖추는 쪽은 그 효과 **또는 그보다 적은** 효과를 적은 `proc` 이거나, 효과가 없으면 `fn` 이다.

2 트레이트 안에서 `self` 는 그것을 갖출 타입 자신을 가리킨다.

3 타입이 `satisfies` 로 트레이트를 적으면, 처리기는 그 목록의 op 이 **정확히 그 시그니처로** 있는지 검사한다. 하나라도 없거나 어긋나면 번역이 거부된다.

4 어긋나는 갈래는 다섯이다.

표 45 — 트레이트를 갖추지 못한 자리

진단	무엇이 어긋났는가
`E-TRAIT-UNDEF`	`satisfies` 가 **선언되지 않은** 트레이트를 부른다 — 아무것도 가리키지 않는 주장은 아무것도 검사하지 아니한다
`E-TRAIT-MISSING`	목록의 op 이 없다
`E-TRAIT-SIG`	op 은 있으나 **매개변수의 수**가 다르다 — 정확히 맞지 않는 트레이트는 계약을 실어 나르지 못한다
`E-TRAIT-EFFECT`	갖춘 쪽이 트레이트가 적지 **않은 효과**를 가진다. 부르는 쪽은 트레이트의 계약을 보고 판단하므로, 그 계약에 없는 일을 하면 판단이 틀린다
`E-TRAIT-RECV`	**op** 이 트레이트를 갖추겠다고 적었다 — 갖추는 것은 **타입**이다

5 트레이트의 op 이 효과 줄에 `via self` 를 적으면, 갖추는 쪽은 그 op 에 **할당 계열 효과** — `alloc` · `heap` · `lock` · `atomic` — 를 트레이트보다 더 적을 수 있다. 그 밖의 효과를 더 적는 것은 여전히 `E-TRAIT-EFFECT` 다. 더 적은 효과는 제네릭 op 이 `via <타입 매개변수>` 로 부르는 쪽까지 나른다(§7.1.1 (6)).

> **쉬운 말로**
>
> 트레이트가 하는 일은 「이 타입은 이런 것들을 할 줄 안다」를 **부르는 쪽이 믿을 수 있게** 만드는 것이다. 그래서 이름만 맞고 효과가 다르면 그 믿음이 깨진다 — 효과까지 맞아야 트레이트가 계약을 실어 나른다.

> **예제 (example) — 트레이트를 갖춘다**
>
> **소스 코드 (source)**

```
module ex_trait .

rem 트레이트는 타입이 갖춰야 할 op 의 목록이다.
trait shape do
  area input s self . output u64 . effects none .
end .

rem `satisfies` 를 적으면 그 목록을 갖췄는지 검사받는다.
struct rect do
  satisfies shape .
  w u8 .
  h u8 .
end .

fn rect.area input s rect . output u64 .
do
  return mul widen u64 field s. w . . widen u64 field s. h . . . .
end .
```

> **거부되는 예제 (rejected) — 갖추겠다고 적고 안 갖추면**
>
> **소스 코드 (source)**

```
module ex_trait_bad .

trait shape do
  area input s self . output u64 . effects none .
end .

struct rect do
  satisfies shape .     rem 갖추겠다고 적었는데
  w u8 .
  h u8 .
end .

rem `rect.area` 를 만들지 않았다
```

진단: `E-TRAIT-MISSING`

예제 (example) — op 을 여럿 가진 트레이트

소스 코드 (source)

```
module ex_trait_many .

rem 서명마다 한 줄 — 이름으로 시작하고 입력 · 출력 · 효과 차례다.
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 .

rem 효과 줄이 없는 서명은 `fn` 으로 갖춘다.
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 lit rect do w add field s. w . k. . . h add field s. h . k. . . end . .
end .

rem 효과를 적은 서명은 그 효과를 적은 `proc` 으로 갖춘다.
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 .
```

거부되는 예제 (rejected) — 효과 줄이 없는 proc 으로 효과 없는 서명을 갖추려 한다

소스 코드 (source)

```
module ex_trait_proc_noeff .

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 .
```

진단: `E-TRAIT-EFFECT`

쉬운 말로

트레이트는 *"*이 타입은 이런 일을 할 수 있다*"*를 **검사받는 형태로** 적는 방법이다. 주석으로 적으면 코드가 바뀌어도 그대로 남지만, `satisfies` 는 갖추지 못하는 순간 번역이 거부된다.

주의 — 상속이 아니다

트레이트는 타입 사이에 **위아래를 만들지 아니한다.** 어떤 타입이 트레이트를 갖췄다는 것은 *"*그 *op* 들을 갖고 있다*"*는 사실일 뿐이며, 다른 타입의 무엇을 물려받지 않는다. 이 언어에는 상속이 없다.

6.11.3 붙은 op 을 부른다 — method

`method` 는 값에 붙은 op 을 부르는 폼이다. 모양은 `method <값> <마디> … <이름> <인자> … .` 이다.

마디와 이름은 **타입을 따라가서** 갈린다. 수신자의 타입에서 마디를 하나씩 내려가다가, 그 타입에 그 이름의 붙은 op 이 있으면 **거기가 이름**이고 그 뒤는 전부 인자다. 그러므로 `method o inner area` 는 `o` 의 `inner` 에 붙은 `area` 를 부르고, `method s plus 5` 는 `s` 에 붙은 `plus` 를 `5` 로 부른다.

수신자는 이름일 필요가 없다. 다른 폼의 결과여도 되며, 그때 수신자의 타입은 그 폼의 출력 타입이다 — 그래서 `method (method x grow 2) area` 가 성립한다.

수신자의 타입을 정할 수 없으면 처리기는 프로그램을 거절한다. 붙은 op 은 **타입으로** 찾는 것이므로, 타입을 모르면 부를 op 도 정해지지 않는다.

예제 (example) — 붙은 op 을 부른다

소스 코드 (source)

```
module ex_method .

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 .

export fn twice_area input s rect . output u64 .
do
  return mul 2 method s. area. . . .
end .
```

참고 (informative)

중위로 적는 표기(`s..area`) 는 **없다**. 그것은 언어에서 유일하게 오른쪽에서 왼쪽으로 읽히는 자리였고, 다단 수신자를 적을 방법이 없었다. 폼으로 통일하면 호출은 **언제나 이름이 먼저**다.

6.11.4 붙은 op 이 어긋나는 자리

1 `method` 는 **수신자와 이름**을 모두 요구한다(E-METHOD-RECV).

2 수신자의 타입에 그 이름의 붙은 op 이 없으면 거부된다(E-METHOD-UNDEF) — `fn <타입>.<이름>` 으로 선언되어 있어야 한다.

주의 —

붙은 op 은 **이름칸을 나누는 장치**이지 상속이 아니다(§6.11.2). 그러므로 없는 이름을 부르면 위로 찾아 올라가는 일이 없고, 그 자리에서 바로 거부된다.

거부되는 예제 (rejected) — 그 타입에 그 이름의 붙은 op 이 없다

소스 코드 (source)

```
module ex_method_undef .

struct p do
  x u8 .
end .

fn f input s p . output u8 .
do
  return method s. nosuch. . .
end .
```

진단: E-METHOD-UNDEF

6.12 한 줄기로 흘리기 — `pipe`

1 `pipe` 는 원천(source)의 원소를 하나씩 꺼내 스테이지 (stage) 를 차례로 지나게 하고, 종결자 (terminal) 하나로 끝내는 문(statement)이다. 모양은 다음과 같다.

문법 틀 (grammar shape) — pipe 문

```
pipe <원천> do
  <스테이지> .        rem 없어도 된다. 여럿일 수 있다.
  <종결자> .          rem 정확히 하나.
end
```

쉬운 말로

슬라이스를 훑는 반복은 거의 언제나 같은 뼈대다 — 인덱스를 두고, 끝인지 보고, 원소를 꺼내 조건을 보고, 무언가를 쌓고, 인덱스를 늘린다. 그 뼈대를 손으로 쓰면 **하려는 일**(«숫자만 남기고 센다»)이 인덱스와 카운터 사이에 묻히고, 경계를 한 칸 틀리는 실수가 거기서 난다. `pipe` 는 뼈대를 언어가 맡고, 사람은 **하려**

는 일만 줄마다 한 낱말로 적게 한다. 그러면서도 손으로 쓴 반복과 똑같이 한 번만 훑고 중간 배열을 만들지 않는다(§6.12.1). 아래 두 op 은 같은 답을 낸다.

예제 (example) — 같은 일을 손으로 쓴 반복과 pipe 로 — 둘 다 한 번 훑는다

소스 코드 (source)

```
module ex_pipe_why .

fn is_digit input c u8 . output bool . do return and ge c. 48 . le c. 57 . . . end .

rem 손으로 쓴 반복 — 카운터·인덱스·조건·증가를 사람이 하나하나 맞춘다.
fn digits_loop input s slice u8 . output u64 .
do
  var n u64 0 .
  var i usize 0 .
  while lt i. len s. . . do
    if is_digit idx s. i. . . do set n. add n. 1 . . end .
    set i. add i. 1 . .
  end .
  return n. .
end .

rem 같은 일을 pipe 로 — «숫자인 것만 남기고, 센다». 하는 일이 줄마다 한 낱말로 보인다.
fn digits_pipe input s slice u8 . output u64 .
do
  return pipe s. do
    filter is_digit .
    count .
  end . .
end .
```

2 스테이지는 일곱이다: `filter` · `map` · `take` · `skip` · `scan` · `zip` · `enumerate`. 종결자는 다섯이다: `collect into` · `fold` · `count` · `any` · `all`. 이 목록 밖의 낱말을 스테이지 자리에 쓰는 것은 적합하지 아니하다(E-PIPE-STAGE).

2a `collect into <자리>` 의 그 자리는 흘러오는 값을 **잃지 않고** 담을 수 있어야 한다. 더 넓은 값을 좁은 그릇에 담는 것은 적합하지 아니하다(E-TYPE-COLLECT) — 값을 잃는 변환은 암묵적으로 일어나지 아니한다(§6.2.5 (1)). 좁히려면 `map` 으로 **적어서** 한다.

2b 각 낱말의 뜻은 다음과 같다. «흐름» 은 원천의 원소가 앞 스테이지를 지나 이 자리에 이르는 차례이고, «op» 은 이름 붙은 op 이다(4). 원소 타입을 `t`, 누산값 타입을 `a` 로 적는다.

- `filter <op>` (스테이지, op 은 `t → bool`) — op 이 참인 원소만 다음으로 보낸다. 거짓인 원소는 버린다.
- `map <op>` (스테이지, op 은 `t → u`) — 원소마다 op 을 불러 그 답을 다음으로 보낸다. 원소 타입이 `u` 로 바뀔 수 있다.
- `take <n>` (스테이지) — 앞의 `n` 개만 보내고 **훑기를 멈춘다**(§6.12.2). 원소가 `n` 보다 적으면 있는 만큼.
- `skip <n>` (스테이지) — 앞의 `n` 개를 버리고 나머지를 보낸다.
- `enumerate <op>` (스테이지, op 은 `(u64, t) → u`) — 순번 `i`(0 부터)와 원소를 op 에 주고 그 답을 보낸다.
- `zip <원천2> <op>` (스테이지, op 은 `(t, t2) → u`) — 원소와 `원천2` 의 같은 자리 원소를 op 에 주고 그 답을 보낸다. **짧은 쪽이 끝나면 흐름이 끝난다.**
- `scan <초기값> <op>` (스테이지, op 은 `(a, t) → a`) — 누산값을 `초기값` 으로 두고, 원소마다 `누산값 = op(누산값, 원소)` 로 고친 **뒤의** 누산값을 보낸다. 초기값 자신은 보내지 아니한다.
- `collect into <자리>` (종결자) — 흐름의 원소를 `<자리>`(`mut slice`)에 앞에서부터 담고, **담은 개수**(`u64`)를 낸다. 자리는 흐름을 모두 담을 만큼 길어야 한다(2c).

- `fold <초기값> <op>` (종결자, op 은 `(a, t) → a`) — `scan` 처럼 누산하되 마지막 누산값 하나(`a`)를 낸다. 흐름이 비면 초기값이다.
- `count` (종결자) — 흐름의 원소 수(`u64`)를 낸다.
- `any <op>` (종결자, op 은 `t → bool`) — op 이 참인 원소가 있으면 참. 흐름이 비면 거짓. 첫 참에서 멈춘다.
- `all <op>` (종결자, op 은 `t → bool`) — 모든 원소에서 op 이 참이면 참. 흐름이 비면 참. 첫 거짓에서 멈춘다.

2c 받는 자리가 차면 **말없이 멈추지 아니한다.** 흐름이 받는 자리보다 긴 것을 번역 시점에 알면 — 두 길이가 머리의 계약(`requires eq len x. . N .`, `array T N` 입력이 그리 된다)에 적혀 있고 사이의 스테이지가 개수를 모르는 것(`filter·zip`)이 아니면 — 번역이 거부한다(`E-COLLECT-FULL`). 알 수 없으면 실행 중, 담지 못할 원소가 오는 순간 받는 자리가 찼다고 말하며 멈춘다(`E-VM-BOUNDS`, 네이티브는 같은 문장의 panic). 들어갈 만큼만 담으려면 `take` 로 **적는다.**

2d `filter·any·all` 의 op 은 **판정**이다 — 출력이 `bool` 이어야 한다. 수를 내는 op 을 주는 것은 적합하지 아니하다(`E-PIPE-PRED`). 0 이 아닌 수를 참으로 읽는 규칙은 이 언어에 없다(§6.2.16 (4)).

> **거부되는 예제 (rejected) — 다섯을 셋 칸에 담는다 — 두 길이를 번역 시점에 안다**
>
> **소스 코드 (source)**
>
> ```
> module ex_collect_full .
>
> fn dbl input a u8 . output u8 . do return wrap_add a. a. . . end .
>
> proc over input xs array 5 u8 . input out mut array 3 u8 . output u64 . effects none . do
>   return pipe xs. do
>     map dbl .
>     collect into out. .
>   end . .
> end .
> ```
>
> 진단: `E-COLLECT-FULL`

> **거부되는 예제 (rejected) — 수를 내는 op 을 판정 자리에 준다**
>
> **소스 코드 (source)**
>
> ```
> module ex_pipe_pred .
>
> fn as_flag input a u8 . output u8 . do return a. . end .
>
> fn nonzero input xs slice u8 . output u64 . do
>   return pipe xs. do
>     filter as_flag .
>     count .
>   end . .
> end .
> ```
>
> 진단: `E-PIPE-PRED`

3 한 `pipe` 문은 종결자를 **정확히 하나** 갖는다. 종결자 없이 끝나거나 둘을 두는 것은 적합하지 아니하다.

4 스테이지의 인자는 **이름 붙은 op 을 가리키는 이름**이다. 이름 없는 함수(람다)는 이 언어에 없으므로, 스테이지 인자로 그 자리에서 만든 함수를 줄 수 없다.

5 스테이지 op 은 원소 하나만 받는다 — 매개변수가 정확히 하나이고 능력(capability)을 받지 아니한다. 그렇지 않은 op 을 스테이지로 쓰는 것은 적합하지 아니하다(`E-FOLD-OP`). 문맥(7)을 적으면 매개변수가 하나 더 있다.

5a `fold` 와 `scan` 의 op 은 매개변수가 둘이며, **첫째가 누산값이고 둘째가 원소**다. 한 걸음은 `누산값 = op(누산값, 원소)` 이므로 첫 매개변수의 타입과 출력 타입은 같아야 한다. 다른 타입으로 적는 것은 적합하지 아니하다(`E-FOLD-ORDER`).

6 `take` 와 `skip` 의 개수는 번역 시점에 정해진 음이 아닌 값이어야 한다(§6.8). 실행 시점에야 알 수 있는 값을 주는 것은 적합하지 아니하다.

7 op 을 부르는 스테이지와 종결자 — `filter·map·any·all·scan·fold·zip·enumerate` — 는 op 이름 뒤에 문맥 (context) `with <식>` 을 둘 수 있다. 문맥은 op 의 **마지막 인자**로 건너간다: `filter·map·any·all` 은 (원소, 문맥), `fold·scan` 은 (누산값, 원소, 문맥), `zip` 은 (원소, 짝 원소, 문맥), `enumerate` 는 (번호, 원소, 문맥)을 받는다. 매개변수 수나 문맥의 타입이 맞지 않거나 문맥 매개변수가 `mut·owned` 이면 적합하지 아니하다(`E-PIPE-CONTEXT-ARG`). op 을 부르지 않는 `take·skip·count·collect` 에 문맥을 적거나, op 이름 뒤에 `with` 아닌 낱말이 오거나, `with` 뒤가 식 하나가 아니면 적합하지 아니하다(`E-PIPE-WITH`). `with` 는 이 자리의 표지일 뿐 낱말(부록 A.1)이 아니다.

7a 문맥은 **참조 없는 복사값**이다. 문맥 매개변수의 타입은 — 구조체의 칸, 열거의 모든 갈래 알맹이(쓰이지 않는 갈래도), 별칭·newtype 의 밑 타입, 배열의 원소, option·result 의 알맹이까지 — 슬라이스, 참조(`ref·mut_ref`), 포인터, 능력(`cap`), region, `owned` 값, actor 를 품지 아니하여야 한다 (`E-PIPE-CONTEXT-TYPE`). 능력을 쓰는 식이 문맥을 **만들** 수는 있으나, 능력을 문맥 **안에** 넣어 건넬 수는 없다. 문맥 매개변수가 배열 타입(`array n t`)이면 그 값은 저장소를 가리키며 건너가므로 문맥이 될 수 없다(`E-PIPE-CONTEXT-TYPE`). 구조체의 칸인 배열은 구조체와 함께 베껴지므로(§6.2.7 (4a)) 그 구조체는 문맥이 될 수 있다.

7b 문맥 식은 훑기 **전에 한 번** 평가되고, 그 값이 매 호출에 같은 값으로 건너간다. op 이 제 사본을 고쳐도 다음 호출은 보관된 값을 받는다. 원천이 비어도 평가한다. 문맥이 있는 `pipe` 는 원천을 평가한 뒤 절 차례대로 각 절의 피연산자(`zip` 의 짝, `scan·fold` 의 초깃값, `with` 의 식, `collect into` 의 받는 자리)를 왼쪽부터 평가한다. 준비 중에 실패하면 뒤의 준비와 훑기는 일어나지 아니한다. 문맥 식의 효과는 바깥 op 의 효과다(§7). 문맥이 없는 `pipe` 의 차례는 전과 같다 — 원천, 종결자의 피연산자, 스테이지 차례.

> **예제 (example) — 실행 중에 정해지는 문턱을 문맥으로**

소스 코드 (source)

```
module ex_with .

fn above input x u8 . input limit u8 . output bool .
do
  return gt x. limit. . .
end .

export fn count_above input xs slice u8 . input base u8 . output u64 .
do
  return pipe xs. do
    filter above with wrap_add base. 1 . .
    count .
  end . .
end .
```

> **거부되는 예제 (rejected) — 문맥은 참조 없는 값이다**

소스 코드 (source)

```
module ex_with_slice .

fn over input x u8 . input s slice u8 . output bool .
```

```
do
  return gt x. idx s. 0 . . .
end .

export fn f input xs slice u8 . output u64 .
do
  return pipe xs. do
    filter over with xs. .
    count .
  end . .
end .
```

진단: `E-PIPE-CONTEXT-TYPE`

거부되는 예제 (rejected) — op 을 부르지 않는 종결자는 문맥을 받지 않는다

소스 코드 (source)

```
module ex_with_count .

fn above input x u8 . input limit u8 . output bool .
do
  return gt x. limit. . .
end .

export fn f input xs slice u8 . output u64 .
do
  return pipe xs. do
    filter above with 3 .
    count with 1 .
  end . .
end .
```

진단: `E-PIPE-WITH`

참고 (informative)

문맥은 람다가 아니다. 스테이지 op 은 여전히 이름 있는 op 이고, 바깥의 이름을 몰래 읽지 않는다 — 문맥으로 건너간 값만 본다. 참조를 막는 까닭은 주소만 복사해 바깥 저장소를 훑기 내내 지켜보는 길을 닫기 위해서다. 큰 표를 건네야 하면 일반 반복문이나 op 호출로 적는다.

6.12.1 융합은 최적화가 아니라 의미다

1 한 `pipe` 문은 **한 번의 훑기(one pass)** 로 실행된다. 스테이지 사이에 중간 모음 (intermediate collection)이 만들어지지 아니한다. 이것은 처리기가 시도해 볼 수 있는 최적화 (optimisation) 가 아니라 이 문의 **정의**다.

2 그러므로 처리기는 중간 모음을 만들지 아니하여야 한다. 만드는 처리기는 적합하지 아니하다.

참고 (informative)

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. 여기서는 융합 못 할 연산이 **스테이지 어휘에 아예 없어서**, 사고로 절벽에 떨어질 길이 없다.

6.12.2 필요한 만큼만 읽는다

1 `any` 는 참을 내는 첫 원소에서, `all` 은 거짓을 내는 첫 원소에서 훑기를 멈춘다. 멈춘 뒤의 원소는 읽히지 아니하며, 그 원소에 대한 스테이지도 실행되지 아니한다.

2 `take n` 은 n 개를 지나보낸 뒤 훑기를 멈춘다.

> **참고 (informative)**
>
> 이 성질이 있어야 끝이 없는 원천을 `pipe` 로 다룰 수 있다. 멈춤이 의미에 적혀 있지 않으면, 끝없는 원천은 프로그램을 멈추지 않게 만든다.

> **예제 (example) — 스테이지 둘과 종결자 하나 — 한 번의 훑기**
>
> **소스 코드 (source)**

```
module ex_pipe .

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 [1,2,3,4,5] → 2 보다 큰 것만 → 두 배 → out 에 담는다. 중간 배열은 생기지 아니한다.
proc fm 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 0 .
end .
```

> **쉬운 말로**
>
> `pipe` 가 문인 까닭은 융합을 **약속으로** 만들기 위해서다. 식이었다면 스테이지를 값으로 떼어 넘길 수 있고, 그러면 어디까지가 한 줄기인지 소스에서 안 보인다. `do … end` 가 그 경계를 눈에 보이게 못박는다.

6.12.3 `pipe` 밖의 `map`·`filter`·`fold` — 라이브러리의 op

1 `pipe` 밖에서 `map` · `filter` · `fold` 는 **기본 연산이 아니다.** 세 이름은 저자가 op · 매개변수 · 지역의 이름으로 지을 수 있다. `pipe` 안의 같은 낱말은 스테이지와 종결자다(§6.12 (2)).

2 한 슬라이스를 다른 슬라이스에 옮겨 담거나 거르거나 하나로 접는 일은 표준 모듈 `collection` 의 op `map` · `filter` · `fold` 가 한다(§9.3). 할 일은 **원소의 타입이 들고 온다** — 원소 타입이 트레이트 `collection.mappable(apply)` · `collection.keepable(keep)` · `collection.foldable(combine)` 을 갖추고(§6.11.2), op 은 그 타입을 번역 시점 인자로 받는다. 갖추지 않은 타입을 주면 번역이 거부된다.

2a `collection.map` 과 `collection.filter` 는 받는 자리(`mut slice`)에 써 넣고 **써 넣은 원소의 수**(`u64`)를 낸다 — `collect into` 와 같다. 받는 자리가 차면 써 넣으려는 순간 실행이 멈춘다.

2b 언어 개정 1.12 까지 기본 연산이던 이름은 여섯이다 — `map` · `filter` · `fold` · `count` · `seg` · `segs`. 개정 1.13 부터 여섯 모두 저자의 이름이다. 정의되지 않은 채 그 이름을 부르면 다른 이름처럼 거부되고(`E-IR-UNDEF`), 처리기는 옮길 곳을 함께 말한다 — `count` 는 `bitset_count` · `stack_count` · `len`(§6.2.12 (4a)), `seg` · `segs` 는 `segview.piece_at` · `segview.piece_count`, 전위 `map` · `filter` · `fold` 는 위 (2) 의 `collection`.

> **예제 (example) — 원소 타입이 할 일을 들고 온다**
>
> **소스 코드 (source)**

```
module ex_collection .

use collection .

struct twice do
```

```
  satisfies collection.mappable .
  v u8 .
end .

fn twice.apply input x twice . output twice . do
  return lit twice do v wrap_add field x. v . field x. v . . . end . .
end .

proc twice_all input xs slice u8 . input out mut slice u8 . output u64 . effects none . do
  return collection.map twice. view_array twice xs. . view_array twice out. . . .
end .
```

실행 결과 (output)

```
twice_all([1,2,3], [0,0,0]) = 3
```

쉬운 말로

이 셋은 한때 기본 연산이었다(`map <받는 자리> <op> <원천>` 꼴). 기본 연산의 이름은 저자의 이름이 될 수 없으므로 흔한 낱말 셋이 막혀 있었다. op 을 값으로 넘기는 길이 이 언어에 없으므로, 라이브러리의 제네릭은 할 일을 원소의 타입에서 찾는다 — 정렬이 비교를 그렇게 찾는 것과 같다. 부를 때마다 그 타입 전용 op 이 만들어지고 할 일은 직접 호출로 박힌다.

쉬운 말로

슬라이스를 옮겨 담거나 거르거나 접는 일은 대개 `collection` 없이 `pipe` 로 적는다(§6.12 (2)). `pipe` 는 위 (2) 의 길이 못 하는 셋을 한다 — op 을 이름으로 고르므로 한 원소 타입에 변환 · 판정이 몇 개든 되고, 실행 중에 정해지는 값을 `with` 로 건네며(§6.12 (7)), `map` 의 op 이 다른 타입을 내면 흐름의 원소 타입이 바뀐다. 원소를 칸 하나짜리 구조체로 감쌀 일도 없다. 한 곳에서만 쓰는 op 은 그 몸 안에 지역 op 으로 둔다(§6.10.5). `collection` 은 할 일을 원소의 타입이 들고 다녀야 하는 일반 코드를 위한 것이다. 옛 전위 `map` · `filter` · `fold` 를 부르면 처리기도 `pipe` 를 먼저 말한다.

예제 (example) — 옮겨 담기는 pipe 로 — 실행 중의 값은 with 로

소스 코드 (source)

```
module ex_pipe_move .

fn add_k input x u8 . input k u8 . output u8 . do
  return wrap_add x. k. . .
end .

proc shifted input xs slice u8 . input k u8 . input out mut slice u8 . output u64 . effects
none . do
  return pipe xs. do
    map add_k with k. .
    collect into out. .
  end . .
end .
```

실행 결과 (output)

```
shifted([1,2,3], 10, [0,0,0]) = 3
```

7 효과와 권한 (Effects and capabilities)

1 §6 이 계산을 정했다면, 이 조항은 그 계산이 **바깥세상과 어떻게 닿는가**를 정한다.

7.1 효과

1 효과 (effect) 는 op 이 바깥세상에 주는 영향이다. 파일을 읽는 것, 메모리를 할당하는 것, 기다리는 것 따위가 각각 효과다.

2 효과는 정해진 **원자**의 집합이다. 원자의 목록은 고정되어 있으며, 저자가 새 원자를 만들 수 없다.

표 46 — 효과 원자

원자	무엇을 뜻하나
`none`	아무 효과도 없다. 이것이 바닥이다
`alloc`	**고정 창**에서 메모리를 얻거나 돌려준다 — 그 창은 자라지 않는다(⟦§8.1⟧)
`heap`	**자라는 뿌리**에서 메모리를 얻는다 — 실행 중에 더 받아 올 수 있다. 운영체제가 있는 기계에만 있다(⟦§8.1⟧)
`io`	바깥과 자료를 주고받는다(파일·연결·표준 입출력)
`wait`	기다린다. 다른 누구의 도움 없이도 언젠가 깨어난다
`concurrent`	완결이 **다른 실행 흐름의 진행에 달려 있다**
`lock`	자물쇠를 잡는다
`atomic`	나눌 수 없는 읽기·쓰기를 한다
`state`	모듈이나 액터가 지닌 상태를 고친다
`panic`	명시적 `panic` 으로 프로그램을 끝낼 수 있다 — 경계 밖 색인·넘침 같은 트랩은 효과가 아니라 계약이 맡는다(⟦§6.4.6⟧)
`device`	장치를 직접 건드린다
`unsafe`	언어가 검사할 수 없는 일을 한다
`page_fault`	메모리 접근이 페이지 부재를 일으킬 수 있다
`blocking`	실행 흐름을 붙잡아 둘 수 있다
`cancel`	취소될 수 있다
`detach`	만든 실행 흐름을 떼어 놓는다

2a 이 가운데 다섯 — `lock` · `page_fault` · `blocking` · `cancel` · `detach` — 은 그 일을 하는 **기본 연산이 아직 없다.** 적을 수 있고 부르는 쪽으로 번져 올라가지만(아래 (4)), 본문이 그 일을 하는지 처리기가 알아낼 길이 없으므로 **적은 대로 믿는다**. 선언만 하고 하지 않아도 거절하지 아니하고((6)), 믿고 넘어갔다는 것을 알린다.

2b `device` 는 장치 레지스터를 만지는 일(`read_volatile` · `write_volatile` 이 장치의 레지스터를 향하는 것, 장치 레지스터 뷰의 칸을 읽고 쓰는 것)과 인터럽트 처리기(`vector` 절을 가진 op)가 낸다. 다른 효과처럼 내면 적어야 하고, 적어 놓고 내지 않으면 알린다((6)).

3 op 은 자기가 내는 효과를 `effects` 절에 적어야 한다. **적은 것보다 많은 효과를 내면 번역이 거부된다.**

4 효과는 부르는 쪽으로 **번져 올라간다.** 어떤 op 을 부르면 그 op 의 효과가 부르는 쪽의 효과에 더해진다.

4a `state` 는 **op 자신이 지닌 상태**(모듈 상태 · 액터의 상태 칸)를 고치는 일을 말한다. 부른 쪽이 건넨 저장소(`mut` · `mut_ref` 자리, `collect into <이름>`)에 쓰는 것은 `state` 가 아니다 — 그 쓰기는 **타입이 이미 밝히고 있고**, 부른 쪽은 자기 이름을 통해 그것을 본다. 감춰진 상태가 아니라서 효과로 셀 것이 없다. 그런 op 이 `effects state` 를 적는 것은 허용하며, 적었다고 「선언만 하고 안 한다」(`W-EFFECT-OVER`)로 세지 아니한다.

5 어떤 효과는 **다른 효과를 딸고 온다.** 다음 표가 그 전부다.

표 47 — 딸려 오는 효과 (닫힘 관계)

적으면	함께 적은 것이 된다	왜
`concurrent`	`wait`	동료의 진행을 기다려야 한다면 중단될 수도 있다

5a 그러므로 선언한 효과 집합 `S` 는 **닫아서** 본다 — `close(S) = S ∪ {wait ¦ concurrent ∈ S}`. 두 집합을 견주는 것도 닫은 뒤다: 요구 `I` 가 선언 `D` 에 담기는 것은 `close(I) ⊆ close(D)` 일 때다. 그래서 `concurrent` 를 적으면 `wait` 을 따로 적지 않아도 되고, `wait` 만 적고 `concurrent` 한 일을 하면 거부된다.

5b 닫는 일은 **한 곳에서만** 일어난다. 원자의 표(위)는 평평하게 남으며, 포함 관계를 그 표에 섞지 아니한다.

> **도해 (diagram) — 효과는 부르는 쪽으로 번져 올라가고, 닫은 뒤에 견준다**

```
 worker  effects concurrent         close = {concurrent, wait}
   ▲ called by boss
 boss    effects state wait         close = {state, wait}              → E-EFFECT (no
concurrent)
 boss    effects state concurrent   close = {state, concurrent, wait}   → ok
```

> **쉬운 말로**

효과 선언이 하는 일을 한 문장으로 말하면 이렇다 — *"이 함수를 부르면 무슨 일이 벌어질 수 있는지가 시그니처에 적혀 있다."* 본문을 열지 않아도, 그 함수가 부르는 다른 함수를 따라가지 않아도 알 수 있다.

6 반대로, **선언해 놓고 내지 않는 효과**도 잘못이다. 처리기는 그것을 알린다. 선언한 효과는 부르는 쪽이 감당해야 하는 비용이므로(순수한 op 은 `io` 를 내는 op 을 부를 수 없다), 내지도 않는 효과를 적으면 이 op 이 하는 일을 **부풀려 말하는 것**이 된다.

7 `fn` 은 순수함이 **계약으로 정해진** op 이므로 효과 집합이 이미 `none` 이다. 그래서 `fn` 에는 `effects` 절을 적지 **않는다**. 적으면 처리기가 프로그램을 거절한다 — 뜻은 같지만 **한 뜻에 두 표기**가 되고, 그러면 읽는 사람도 도구도 두 가지를 알아야 한다. `proc` 은 다르다: 거기서 `effects` 절은 **좁히는** 일을 한다. `proc` 은 그 절을 **반드시** 적는다 — 없으면 `E-EFFECT-MISSING` 이다(효과가 없으면 `effects none .`). 절이 없을 때의 뜻을 정해 두면 그 뜻이 읽는 곳마다 갈렸다.

> **예제 (example) — 효과 선언**

소스 코드 (source)

```
module ex_effect .

rem 순수한 op — 바깥세상을 건드리지 않는다. `fn` 이 곧 그 선언이다.
export fn double input n u32 . output u32 .
  requires le n. 1000 . .
do
  return mul n. 2 . .
end .
```

```
rem 효과를 내는 op — 무슨 효과인지 계약에 적는다.
export proc note_and_add input n u32 . output u32 .
  effects panic .
  requires le n. 1000 . .
do
  if gt n. 500 . do panic . end .
  return add n. 1 . .
end .
```

주의 — 효과 선언은 문서가 아니라 검사다

주석으로 "이 함수는 파일을 읽습니다"라고 적는 것과는 다르다. 주석은 코드가 바뀌어도 그대로 남지만, 효과 선언은 본문이 더 많은 일을 하는 순간 **번역이 거부된다**. 그리고 **덜 하는 것도** 알려진다 — 선언과 본문이 어느 쪽으로든 어긋나면 그 절은 거짓말이다.

7.1.1 효과 줄을 적는 법

1 효과 줄은 **집합**이지 목록이 아니다. 같은 낱말을 두 번 적는 것은 거부된다 (E-EFFECT-DUP) — 두 번째는 첫 번째가 하지 않은 말을 하나도 하지 아니한다.

2 목록에 없는 낱말은 거부된다(E-EFFECT-UNDEF). 효과의 이름은 §7.1 의 닫힌 집합에서 온다.

3 `none` 은 **다른 것과 함께 올 수 없다**(E-EFFECT-NONE-MIX). 효과가 있으면서 없을 수는 없다 — 효과를 적든지, 없다고 적든지 하나다.

4 계약으로 이미 순수한 op 에 `effects none .` 을 덧붙이는 것은 거부된다 (E-EFFECT-REDUNDANT). 한 뜻은 **한 철자**를 갖는다.

5 순수한 op 은 효과를 **선언할 수 없다**(E-EFFECT-CALC). 순수함과 효과 있음은 갈래가 다르며, 갈래를 고르는 것은 낱말이지 절이 아니다.

6 효과 줄에 `via <타입>` 을 적을 수 있다. 그 뜻은 «그 타입의 op 들이 적은 **할당 계열 효과** (`alloc` · `heap` · `atomic`)도 이 op 의 선언이다» 이다. 제네릭 op 이 타입 매개변수를 적으면 (`effects state via a .`) 단형화된 인스턴스마다 그 타입이 정한다 — 빌린 바이트 위의 범프로 열면 `state` 뿐이고, 자라는 뿌리를 딛는 얼로케이터로 열면 인스턴스의 선언에 `heap` 이 선다. `via` 로 들어온 효과는 이 op 이 그 가운데 일부만 내어도 «선언만 하고 안 하는 효과» 로 치지 아니한다.

쉬운 말로

`via` 가 없으면 제네릭 컨테이너는 둘 중 하나를 해야 한다 — 모든 얼로케이터가 할 수 있는 가장 큰 효과를 늘 적거나(범프로 쓸 때 거짓이다), 가장 작은 것만 적거나(힙으로 쓸 때 **숨은 할당**이다). `via` 는 그 선택을 **타입 인자가 정하게** 한다.

쉬운 말로

(1)·(4) 가 막는 것은 틀린 프로그램이 아니라 **같은 것을 말하는 두 번째 방법**이다. 두 방법이 생기면 읽는 사람은 둘의 차이를 찾게 되고, 차이가 없다는 것을 알아내는 데 시간을 쓴다. 그리고 언젠가 한쪽만 고쳐진다.

거부되는 예제 (rejected) — 순수한 op 은 효과를 선언할 수 없다

소스 코드 (source)

```
module ex_calc_eff .

fn f output u8 . effects io .     rem 효과가 있으면 `proc` 이다
do
```

```
  return 1 .
end .
```

진단: `E-EFFECT-CALC`

7.2 권한

1 효과를 내려면 그것을 낼 자격이 있어야 한다. 그 자격을 나타내는 값이 권한 (capability) 이다.

2 권한은 **인자로 건네받는다.** 아무 데서나 꺼내 쓸 수 있는 전역 권한이 없다.

3 권한에는 종류가 있다. 어떤 종류인지가 타입에 적혀 있으며, 종류가 맞지 않으면 번역이 거부된다. 권한을 요구하는 자리는 **종류로** 인가한다 — 권한 값이 있다는 사실만으로는 되지 아니한다.

표 48 — 권한의 종류

종류	그것으로 할 수 있는 일
`cap io`	표준 입출력 — 읽고 쓴다
`cap clock`	시각을 묻는다
`cap random`	운영체제의 엔트로피를 읽는다
`cap net`	연결을 만들고 주고받는다 — **이름을 주소로 바꾸는 일도 여기 든다**
`cap file_system`	파일과 디렉터리를 연다·읽는다·쓴다
`cap process`	다른 프로그램을 띄우고, 그 출력을 읽고, 끝나기를 기다리거나 끝낸다
`cap tty`	터미널의 크기를 묻고 모드를 바꾼다
`cap allocator`	고정 창에서 메모리를 얻는다(자라지 않는다)
`cap heap`	자라는 뿌리에서 메모리를 얻는다 — 운영체제가 있는 기계에서만 받을 수 있다
`cap env`	실행 환경의 변수를 읽는다
`cap c`	씨(C) 함수를 부른다(§6.9)
`cap mmio`	장치 레지스터를 읽고 쓴다
`cap args`	프로그램을 부를 때 준 인자를 센다·읽는다
`cap machine`	기계 명령을 직접 낸다(⟦§6.9⟧) — 처리기가 그 안을 보지 못한다

3a 권한의 **이름**은 닫혀 있지 아니하다 — 저자가 자기 권한을 지어 매개변수로 받을 수 있다. 닫혀 있는 것은 **처리기가 뜻을 정해 둔 종류**이며, 그것은 위 표의 열이다.

3b 처리기가 뜻을 정해 둔 일(시각을 묻기·연결을 열기 따위)을 하려면 **그 일이 요구하는 종류**의 권한을 건네야 한다. 어긋나면 다음 둘 중 하나로 거부된다.

표 49 — 권한이 어긋나는 두 모양

무엇을 건넸나	무슨 일이 일어나나
다른 종류의 권한 (`cap io` 를 시계에)	종류가 다르다고 거부된다
처리기가 **모르는 이름**의 권한	그 권한으로는 그 효과를 낼 수 없다고 거부된다

> **참고 (informative)**
>
> 종류를 나누는 까닭은 **최소 권한**이다. 권한이 하나뿐이면 파일을 읽으려고 받은 자격이 연결도 열 수 있게 되고, 그러면 op 의 서명이 *"이 op 이 무엇을 할 수 있는가"*를 더는 말해 주지 못한다. 종류가 나뉘어 있으므로 서명만 읽고도 닿을 수 있는 바깥세상의 넓이를 안다.

> **도해 (diagram) — 권한은 시작점에서 받아 인자로만 내려간다**
>
> ```
> environment ──► main  input out cap io .  input fs cap file_system .
>                  ├──► save fs     holds file_system only → can open files, cannot print
>                  └──► show out    holds io only          → can print, cannot open files
> an op handed no capability cannot have that effect — there is no global way to get one
> ```

4 프로그램의 시작점은 실행 환경에서 권한을 받는다(§5.5). 시작점이 받지 않은 권한은 프로그램 어디에도 없다.

5 권한은 **번역 시점의 표시**이며 실행 중의 값이 아니다. 그러므로 권한을 받는 op 도 그 때문에 실행 중 인자를 하나 더 받지 아니한다.

> **쉬운 말로**
>
> 곧 `input k cap io .` 을 적는 값은 **공짜**다. 권한을 받는 op 과 안 받는 op 이 씨(C) 경계에서 **같은 서명**을 갖는다 — 권한은 *"이 op 을 부를 자격이 있는가"*를 번역할 때 묻고, 실행할 때는 물을 것이 남지 않기 때문이다. 경계를 지키는 값이 실행 중 비용으로 돌아오지 않는다는 것이 이 설계의 요점이다. "안전하게 하려면 느려진다"는 여기서 참이 아니다.

6 **효과와 권한은 짝이다.** 효과 줄은 op 이 무엇을 하는지를 적고, 권한 입력은 누가 그것을 허락했는지를 적는다. 같은 일을 두 쪽에서 적는 것이며, 서로를 검사한다:

- 효과 줄에 적은 일은 그것을 허락하는 권한을 입력으로 받아야 한다(없으면 거부된다).
- 권한으로 여는 일을 본문이 하면 그 효과를 효과 줄에 적어야 한다(없으면 `E-EFFECT`).
- 권한을 받아 두는 것만으로 효과가 생기지는 아니한다 — 받고 안 쓰는 권한은 일을 하지 않는다.

표 50 — 효과와 그것을 허락하는 권한

효과	허락하는 권한	없을 때
`io`	`cap io` · `cap file_system` · `cap net` · `cap tty` · `cap clock` · `cap random`	`E-EFFECT-NO-CAP`
`alloc`	`cap allocator` (또는 어휘 `region` 블록)	`E-ALLOC-NOCAP`
`heap`	`cap heap`	`E-HEAP-NOCAP`
`atomic`	`cap atomic`	`E-ATOMIC-NOCAP`
`device`	`cap mmio` — 레지스터를 실제로 만지는 자리에서	`E-MMIO-NOCAP`
`state` · `panic` · `wait` · `concurrent`	— 권한 없이 적는다	—

> **쉬운 말로**
>
> 짝으로 두는 까닭은 서명 한 줄로 두 물음에 답하기 위해서다. `effects io` 만 보면 "바깥과 대화한다"는 것은 알지만 **무엇과** 대화하는지는 모른다. `input fs cap file_system .` 이 그것을 좁혀 준다 — 파일은 열 수 있으나 연결은 열 수 없다. 거꾸로 권한만 있고 효과 줄이 없으면, 부르는 쪽은 그 op 이 순수한지 알 수 없다.

예제 (example) — 권한을 받아야 낼 수 있다

소스 코드 (source)

```
module ex_io .

use stdio .

rem 출력하려면 `cap io` 를 인자로 받아야 한다.
proc main input out cap io . output u8 . effects io .
do
  let n usize stdio.write out. 1 "hello\n" . .
  return cast u8 n. . .
end .
```

거부되는 예제 (rejected) — 권한 없이는 한 바이트도 낼 수 없다

소스 코드 (source)

```
module ex_io_bad .

use stdio .

proc main output u8 . effects io .
do
  let n u64 stdio.write 1 "hello\n" . .   rem 권한을 안 받았다
  return narrow u8 n. . .
end .
```

진단: `E-EFFECT-NO-CAP`

쉬운 말로

두 예제의 차이는 인자 하나뿐이다. 앞의 것은 출력 권한을 받았고 뒤의 것은 받지 않았다. **숨은 표준 출력이 없다** — 어떤 함수도 권한 없이 바깥에 바이트를 낼 수 없다.

6a 권한을 다시 넘길 때 특별한 표기는 없다. 받은 이름을 다른 인자와 똑같이 적는다.

7 권한을 받지 않은 op 은 권한을 요구하는 op 을 부를 수 없다. 넘길 것이 없기 때문이다.

예제 (example) — 권한은 사슬을 타고 내려간다

소스 코드 (source)

```
module ex_cap_chain .

use stdio .

rem 권한을 인자로 받는다 — 이름은 `k`, 타입은 `cap io` 다.
proc say input k cap io . input msg slice u8 . output usize . effects io .
do
  return stdio.write k. 1 msg. . .
end .

rem 부르는 쪽은 자기가 받은 `k` 를 **그냥 이름으로 넘긴다**.
proc say_twice input k cap io . input msg slice u8 . output usize . effects io .
do
  let a usize say k. msg. . .
  let b usize say k. msg. . .
  return add a. b. . .
```

```
end .

rem 시작점은 권한을 **바깥에서** 받는다 — 아무도 스스로 만들지 못한다.
proc main input k cap io . output u8 . effects io .
do
  let n usize say_twice k. "hi\n" . .
  guard eq n. 6 . else return 1 . .
  return 0 .
end .
```

쉬운 말로

사슬이 이렇게 생긴다 — 바깥 → `main` 의 `k` → `say_twice` 의 `k` → `say` 의 `k` → `stdio.write` → 빌트인 연산 `write_out`. 권한을 안 받은 op 은 `say` 를 부를 수가 없다. **넘길 것이 없기 때문이다.** 곧 경계를 지키는 것이 문법이 아니라 **인자**다. 규칙을 외울 것이 없고, 부르는 자리를 보면 자격이 어디서 왔는지 눈으로 따라갈 수 있다.

참고 (informative)

이 규칙의 값어치는 "이 프로그램이 파일을 건드릴 수 있는가"를 **시작점만 보고** 답할 수 있다는 데 있다. 깊은 곳의 어떤 함수가 몰래 파일을 여는 일이 있을 수 없다 — 권한을 받지 않았다면 열 수 없고, 받았다면 그 사슬이 시작점까지 이어져 보인다.

쉬운 말로

열쇠에 비유할 수 있다. 방에 들어가려면 열쇠가 있어야 하고, 열쇠는 누군가 건네줘야 생긴다. 아무도 안 줬는데 열쇠를 지어낼 수는 없다.

7.3 순수함이 뜻하는 것

`effects none` 인 op 은 순수 (pure) 하다. 순수한 op 은 다음을 만족한다.
가) 같은 입력에 언제나 같은 결과를 낸다.
나) 부르는 순서를 바꾸어도 프로그램의 결과가 달라지지 않는다.
다) 결과를 쓰지 않는다면 부르지 않아도 된다.
순수한 op 은 순수하지 않은 op 을 부를 수 없다.
`fn` 이 `mut` 으로 받은 저장소에 쓰면 그 쓰기는 부른 쪽에 보이므로 순수하지 않다(`E-EFFECT-PURITY`). 직접 쓰든, 그 저장소를 보는 지역 이름(`subslice` · `view_array` 로 다시 묶은 것)으로 쓰든, 그것을 `proc` 의 `mut` 입력에 넘기든 같다 — 넘긴 쪽이 쓰면 부른 쪽의 저장소가 바뀐다. `fn` 의 `mut` 입력에 넘기는 것은 그 `fn` 이 같은 규칙으로 검사받으므로 여기서 세지 않는다. op 안에서 지은 저장소에 쓰는 것은 밖에서 보이지 않으므로 순수함을 깨지 않는다.

거부되는 예제 (rejected) — `fn` 이 받은 `mut` 조각을 `proc` 에 맡겨 쓴다

소스 코드 (source)

```
module ex_purity_via .

proc put input out mut slice u8 . output u64 . effects none . do
  set idx out. 0 . 7 .
  return 0 .
end .

fn f input out mut slice u8 . output u64 . do
```

```
  return put out. . .     rem 부른 쪽의 조각이 바뀐다
end .
```

진단: E-EFFECT-PURITY

7.4 다른 언어와 만나는 자리

1 로우엔트는 C 와 주고받을 수 있다. 그 경계는 extern 으로 표시한다.

2 경계를 넘는 자리에서는 언어의 보장이 저절로 이어지지 아니한다. 그래서 경계에는 **계약을 적어야** 하며, 처리기는 그 계약을 경계에서 강제한다.

3 C 쪽으로 나가는 이름은 저자가 정한다. 처리기가 이름을 지어내지 아니한다 — 그 이름은 다른 언어에 하는 약속이므로 약속한 사람이 적어야 한다.

4 씨 쪽의 이름은 link "<이름>" . 절로 적는다. 이것이 (3) 이 말하는 «저자가 정하는 이름» 이며, 처리기는 op 의 이름에서 그것을 지어내지 아니한다.

5 variadic . 절을 붙인 extern 은 고정된 인자 뒤에 **개수가 정해지지 않은 인자**를 더 받는다. 그런 인자에는 계약이 닿지 아니하므로, 그 자리는 부르는 쪽이 책임진다.

6 씨가 **로우엔트를 부르게** 할 수도 있다. export extern 한 op 의 주소를 unsafe_fn <op> 으로 얻어 값으로 넘긴다.

7 그 자리에서 **경계의 방향이 뒤집힌다.** 넘기는 일은 언제·몇 번·어느 흐름에서 불릴지 이 언어가 알 수 없으므로 unsafe 이지만, **불려 들어오는 자리**는 이쪽 문이므로 그 op 의 계약이 인자에 강제된다. 곧 문 안은 이 언어가, 문 밖은 씨가 책임진다.

> **주의 — 경계는 구멍이 아니라 문이다**
>
> 다른 언어와 만나는 자리를 "여기서부터는 아무 보장이 없다"로 두면, 그 순간부터 프로그램 전체의 보장이 무의미해진다. 로우엔트는 그 자리에 **계약을 세워** 무엇을 약속받고 무엇을 약속하는지 적게 한다.

7.5 원자적으로 하는 일

1 atomic 효과가 붙은 연산은 **쪼개지지 아니한다** — 다른 흐름이 그 중간을 볼 수 없다. 읽기·쓰기·더하기·비트 연산·바꿔 끼우기(swap)·견주어 바꾸기(compare-and-swap)가 있다.

1a 값을 내는 원자 연산의 결과는 그 칸의 원소 타입이다. 읽기는 칸의 값을, 더하기 · 빼기 · 비트 연산 · 바꿔 끼우기는 **바꾸기 전의** 값을 낸다. 견주어 바꾸기(atomic_cas)는 바꿨으면 1 을, 아니면 0 을 그 타입으로 낸다.

1b 기계마다 잠금 없이 하는 원자 연산의 폭이 정해져 있다 — 읽기·쓰기의 폭과 읽고-쓰기(더하기 · 빼기 · 비트 연산 · 바꿔 끼우기 · 견주어 바꾸기)의 폭. 칸의 원소가 그보다 넓으면 거부된다(E-ATOMIC-TARGET). 이 처리기에서 cortex_m 은 둘 다 4 바이트, cortex_m0 은 읽기·쓰기 4 바이트 · 읽고-쓰기 없음이고, 그 밖의 기계는 8 바이트까지다. 처리기는 기계마다 그 수를 보여 준다.

2 atomic_fence 는 값을 셈하지 아니하고 **차례만** 정한다 — 그 앞의 일이 뒤의 일보다 먼저 보이게 한다.

3 원자 연산은 그 효과를 계약에 적어야 한다. 적지 아니하고 쓰는 것은 적합하지 아니하다.

4 원자 연산 뒤에 order <이름> 을 붙여 **다른 흐름에게 무엇이 언제 보이는지**를 정할 수 있다. 이름은 다음 다섯이 전부이며, 그 밖은 거부된다(E-ATOMIC-ORDER).

표 51 — 기억 차례 (memory ordering)

이름	뜻
relaxed	원자성만 있고 **차례는 없다**. 값이 찢어지지 않는다는 것뿐
acquire	이 읽기 **뒤의** 일이 앞으로 새어 나가지 아니한다
release	이 쓰기 **앞의** 일이 뒤로 새어 나가지 아니한다

`acq_rel`	둘 다. 읽고 쓰는 연산에 쓴다
`seq_cst`	모든 흐름이 **같은 하나의 차례**를 본다. 가장 강하다

5 차례를 적지 아니하면 `seq_cst` 로 본다. 가장 강한 것이 기본값인 까닭은 그것이 가장 추론하기 쉽기 때문이다 — 약하게 하는 것은 **적은 사람이 그 값을 알고** 하는 일이다.

6 연산에 따라 쓸 수 있는 차례가 다르다. **뜻이 없는 조합은 거부된다**(`E-ATOMIC-ORDER`).

표 52 — 연산마다 쓸 수 있는 차례

연산	쓸 수 있는 차례
읽기(`atomic_load`)	`relaxed` · `acquire` · `seq_cst` — 읽기는 `release` 일 수 없다
쓰기(`atomic_store`)	`relaxed` · `release` · `seq_cst` — 쓰기는 `acquire` 일 수 없다
울타리(`atomic_fence`)	`acquire` · `release` · `acq_rel` · `seq_cst` — 아무 차례도 없는 울타리는 울타리가 아니므로 `relaxed` 는 쓸 수 없다

> **쉬운 말로**
>
> 왜 조합까지 막는가. `atomic_load … order release` 는 **문장으로는 말이 되지만 뜻이 없다** — 읽기가 무엇을 내보낸단 말인가. C 는 그것을 정의되지 않은 행동으로 두었고, 정의되지 않은 행동은 조용하다. 이 언어는 조용한 것을 두지 아니한다.

> **참고 (informative)**
>
> 원자성은 **공짜가 아니다.** 한 흐름만 만지는 값에 원자 연산을 쓰면 그 값은 느려지기만 한다. 그래서 이 언어는 그것을 기본으로 두지 아니하고 **이름으로 고르게** 한다 — 비용이 서명에 적히도록.

7.6 장치와 이야기하는 자리

1 장치 레지스터를 읽고 쓰는 것은 보통의 메모리 접근이 아니다. `read_volatile` 과 `write_volatile` 로 적으며, 처리기는 그 접근을 **합치거나 없애거나 차례를 바꾸지 아니하여야** 한다.

2 이 접근에는 `device` 효과와 그에 맞는 권한이 있어야 한다(§7.2).

3 그 자리를 **보통의 칸처럼** 읽거나 쓰는 것은 거부된다(`E-MMIO-PLAIN`) — 곧 `field <묶음> <레지스터>` 로 읽거나 `set field <묶음>. <레지스터> . <값> .` 으로 쓰는 것이다. 보통의 칸 접근은 처리기가 **합치거나 없애도 되는** 연산이므로, 그 철자로 적힌 장치 접근은 (1) 이 요구하는 것을 **지킬 수 없다.** 같은 뜻을 적는 길이 둘인데 한쪽만 그 약속을 지킨다면, 다른 쪽은 길이 아니라 함정이다.

> **거부되는 예제 (rejected) — 장치 레지스터를 보통의 칸처럼 만진다**
>
> **소스 코드 (source)**
>
> ```
> module ex_mmio_plain .
>
> build tier t1 .
>
> struct gpio do
>   mmio 0x40020000 .
>   moder u32 rw .
> end .
>
> unsafe proc drive input dev cap mmio . input regs mut slice u8 . output u32 . effects device
> unsafe .
> do
>   let g mut view gpio view gpio regs. . .
> ```

```
  set field g. moder . 2 .       rem 보통 칸 쓰기 — 처리기가 없애도 되는 연산이다
  return 0 .
end .
```

진단: `E-MMIO-PLAIN`

> **참고 (informative)**
>
> 보통의 읽기는 "값을 안다"가 목적이므로 처리기가 두 번째 읽기를 지워도 된다. 장치 레지스터는 **읽는 행위 자체가 일**이다 — 읽으면 상태가 바뀌기도 한다. 그래서 지우면 안 되고, 그 사실을 언어가 알아야 한다.

7.6.1 레지스터 묶음 — `mmio`

1 장치의 레지스터들은 짜임으로 적고, 그 짜임에 `mmio <시작 주소> .` 을 붙여 **장치의 지도**임을 밝힌다. 시작 주소는 정수여야 한다(`E-MMIO-BASE`) — 지도에는 시작이 있다.

2 지도 없이 그 자리를 여는 것은 거부된다(`E-MMIO-NOBASE`).

3 레지스터 묶음의 주소는 **권한에서 온다.** 그 자리에 닿는 일은 `cap mmio` 를 가진 op 만 하며(§7.2), 시작 주소는 그 권한이 허락한 범위 안의 주소로 읽는다. 포인터가 능력인 기계(§5.3 (4a))에서는 묶음의 포인터가 진입점이 받은 권한의 능력에서 **파생**된다 — 정수 주소만으로 포인터를 만들지 아니한다. 그런 능력이 없는 기계에서는 권한이 표지일 뿐이고 뜻은 같다.

표 53 — 레지스터 묶음이 거부되는 자리

진단	무엇이 어긋났는가
`E-MMIO-BASE`	시작 주소가 정수가 아니다
`E-MMIO-NOBASE`	`mmio` 를 선언하지 않은 짜임을 자기 주소로 열려 한다
`E-MMIO-FIELD`	레지스터가 기계가 한 번에 닿을 수 있는 모양이 아니다 — 폭이 1·2·4·8 이고 제 자리에 놓여야 한다
`E-MMIO-BYVALUE`	레지스터 묶음을 **값으로** 받으려 한다. 값으로 받는 것은 베끼는 것이고, **장치를 베낄 수는 없다**
`E-MMIO-PERM`	읽기 전용(`ro`)인 레지스터에 쓰려 한다
`E-MMIO-PLAIN`	레지스터를 보통의 칸처럼 읽거나 쓴다 — `read_volatile`·`write_volatile` 로 적어야 한다
`E-MMIO-NOCAP` · `E-MMIO-NOEFFECT`	권한(`cap mmio`)이나 효과(`device`)가 없다
`E-MMIO-NOHOST`	운영체제가 있는 기계에서 절대 주소를 열려 한다 — 그 주소는 장치가 아니다

> **쉬운 말로**
>
> 장치를 값으로 받을 수 없다는 것이 이 절에서 가장 자주 걸리는 자리다. 짜임을 값으로 넘기면 그것은 **베낌**인데, 베낀 것에 쓰는 일은 장치에 닿지 아니한다. 그렇게 쓰인 프로그램은 아무 일도 하지 않으면서 아무 말도 하지 않는다 — 그래서 번역할 때 막는다.

7.7 인터럽트를 받는 자리

1 op 에 `vector <번호> .` 절을 붙이면 그 op 은 **인터럽트 처리기**가 된다. 급함을 함께 적으려면 `priority <수> .` 를 붙인다. 새 낱말을 두지 아니한다 — 절이 그 일을 한다.

2 인터럽트 처리기는 **기계가 부른다.** 그러므로 다음 넷을 지켜야 한다.

표 54 — 인터럽트 처리기가 지켜야 하는 것

지킬 것	어기면	왜
아무도 부르지 아니한다	E-ISR-CALLED	우리 코드가 부르면 **틀린 스택·틀린 우선순위**에서, 인터럽트가 잘못된 상태로 돈다
매개변수가 없다	E-ISR-PARAMS	기계는 인자를 건네지 아니한다
돌려주는 것이 없다	E-ISR-OUTPUT	돌려줄 상대가 없다 — 기계가 깨웠고, 기계가 끊긴 것을 다시 이어 간다
`effects device` 를 적는다	E-ISR-EFFECT	장치 때문에 있는 자리이기 때문이다

3 나누어 쓰는 상태는 인터럽트 처리기와 보통 코드 사이에서 **우선순위와 큐의 규율**로 오간다. 그 규율은 이 문서가 정하지 아니하며, 라이브러리의 일이다(§9.5).

7.2.1 권한이 어긋나는 자리

1 권한은 **종류로** 따진다. 다른 종류의 권한을 받는 것으로는 그 문이 열리지 아니한다 (E-CAP-KIND). 있음이 아니라 **무엇인가**가 권한을 만든다.

1a 권한을 요구하는 기본 연산과 빌트인 연산(입출력·터미널·시계·난수·파일·네트워크·환경·인자)은 그 권한을 **첫 피연산자로 적는다.** 적지 않으면 적합하지 아니하다(E-CAP-MISSING) — 권한을 받아 두는 것만으로는 쓰지 못하고, 이름 없이 권한에 닿는 길은 없다.

1b 그 자리에 적는 이름은 **이 op 이 입력으로 받은 권한의 이름**이다. 권한은 값처럼 옮겨 담는 것이 아니라 머리 줄에서 머리 줄로 건네는 것이다(§7.2). 그러므로 권한을 **지역 이름에 묶는 줄** (`let` · `var` 로 `cap <종류>` 를 적은 것)은 그 자체로 적합하지 아니하다(E-CAP-LOCAL) — 그 이름을 쓰는 자리가 있든 없든 같다.

1c 권한을 입력으로 받는 **op 을 부르는 자리**도 같다. 그 op 의 권한 자리에 적는 이름은 부르는 op 이 입력으로 받은 권한의 이름이어야 하고(E-CAP-LOCAL), 종류가 같아야 하며(E-CAP-KIND), 첫 입력이 권한인 op 을 권한 없이 부를 수 없다(E-CAP-MISSING). 권한을 요구하는 빌트인 연산은 표준 라이브러리가 감싸 내놓으므로(§9.3 (1a)), 응용 프로그램이 이 규칙을 만나는 자리는 대개 그 감싸개를 부르는 줄이다.

1d op 은 권한을 **내놓지 아니한다**. `output` 절에 `cap <종류>` 를 적은 op 은 — 그대로든, `option` 같은 다른 타입의 안에 적었든 — 적합하지 아니하다(E-CAP-OUTPUT). 권한을 돌려주는 op 은 부르는 쪽의 머리 줄에 보이지 않는 둘째 출처가 된다. 권한 칸을 가진 actor 를 띄우는 일은 §8.6 이 따로 정한다(E-CAP-FORGE).

> **거부되는 예제 (rejected) — 받은 권한을 지역 이름에 묶는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_cap_local .
>
> proc main input out cap io . output u8 . effects none .
> do
>   let k cap io out. .   rem 쓰지 않아도 이 줄이 거부된다
>   return 0 .
> end .
> ```
>
> 진단: E-CAP-LOCAL

> **거부되는 예제 (rejected) — 권한을 내놓는 op**
>
> **소스 코드 (source)**

```
module ex_cap_output .

fn screen input out cap io . output cap io .
do
  return out. .
end .
```

진단: `E-CAP-OUTPUT`

2 지금 짓는 기계가 줄 수 없는 권한을 요구하면 거부된다(`E-CAP-NOHOST`).

3 `alloc` 효과를 적고 **얻는 권한을 받지 않는** op 은 거부된다(`E-ALLOC-NOCAP`). 무더기를 쓰겠다고 적었는데 그 자리를 어디서 받는지 적지 않은 것이다. 권한은 `cap allocator` 입력, 영역 매개변수, 또는 `heap` 이 아닌 종류의 영역 블록이다.

3a `heap` 효과도 같다: `cap heap` 입력이나 `region <이름> heap` 블록 없이 적으면 거부된다 (`E-HEAP-NOCAP`). `cap allocator` 는 고정 창의 권한이며 자라는 일을 인가하지 아니한다.

3b 운영체제가 없는 기계(`machine.no_heap`)에서는 자라는 뿌리를 **어느 자리로도** 청할 수 없다 — `heap` 효과, `cap heap` 입력, `region <이름> heap` 블록이 모두 거부된다(`E-HEAP-NOHOST`). 같은 기계에서 고정 창을 깎는 `alloc` 은 쓸 수 있다.

4 `unsafe` 로 표시하고 `unsafe` 효과를 적지 않은 op 은 거부된다(`E-UNSAFE-UNUSED`) — 표시는 부르는 쪽까지 가야 뜻이 있고, 거기까지 실어 나르는 것이 효과 줄이다. 거꾸로 효과 없이 그런 일을 하면 거부된다(`E-UNSAFE-UNDECLARED`).

4a `unsafe` 는 부름을 따라 **끝까지 번진다**. 멈추는 자리는 하나뿐이다: `absorbs machine <이름> .` 절을 적은 op 이다. 그 op 은 몸 안에서 `unsafe` 인 것을 부를 수 있고, **자기 효과 줄에는 그것을 싣지 아니한다**. 그리고 `<이름>` 은 그 몸 안에서 `cap machine` 이다 — 그 권한이 태어나는 자리가 거기다. 다른 종류는 흡수하지 못한다(`E-ABSORB-SCOPE`): 세상에 닿는 권한을 만들어 내면 부르는 쪽이 못 보는 권위가 생긴다.

4b 흡수하는 op 은 다음을 **모두** 갖추어야 하며, 하나라도 빠지면 거부된다.

표 55 — 흡수가 요구하는 것

갖출 것	진단	왜
효과 줄이 `none`	`E-ABSORB-IMPURE`	흡수는 `unsafe` 하나만 멈춘다 — 다른 효과는 부르는 쪽이 알아야 한다
`reference <op> .`	`E-ABSORB-NOREF`	같은 답을 내는 **순수한 판**이 없으면 «옳음» 과 «일관되게 틀림» 을 가를 길이 없다
`requires` 하나 이상	`E-ABSORB-NOCONTRACT`	안은 못 보지만 **문은 볼 수 있다** — 길이·정렬·범위를 문에서 검사한다
`why "…" .`	`E-ABSORB-NOWHY`	무엇을 가정했는지 적히지 않은 약속은 아무도 확인할 수 없다
매니페스트의 허락	`E-ABSORB-PLACE`	**누가** 보증해도 되는지는 그 권리를 원하는 소스가 아니라 프로젝트의 정체 파일이 정한다

4b′ 매니페스트는 흡수를 허락한 모듈을 이름으로 적는다. 적히지 않은 모듈이 흡수하면 거부되며, 매니페스트가 아예 없으면 **아무도** 흡수하지 못한다 — 기본값은 허락 없음이다.

4c 처리기는 흡수한 자리를 **말할 수 있어야** 한다 — 어느 op 이 무엇을 흡수했고 그 근거가 무엇인지 한 줄씩 낸다. 못 보는 구역은 숨기는 것이 아니라 **보이게** 둔다.

4d 흡수는 **선택**이며 의무가 아니다. 흡수하지 않은 op 은 지금처럼 `unsafe` 를 그대로 내보낸다.

> **쉬운 말로**
>
> 번짐은 옳다 — 처리기가 못 보는 것을 부르는 쪽이 모르면 효과 줄이 거짓말이 된다. 그런데 멈추는 자리가 없으면 그 문은 **아무도 못 쓰는 문**이 된다(2026-09-23 실측: 표준 라이브러리의 `asm` 사용 0 건). 그래서 멈추는 자리를 **좁게, 검사되게** 연다. 그 자리는 도구가 목록으로 내고 장부가 서명과 몸의 해시를 든다 — 사람이 지키기로 한 것은 세어지거나 사라진다.

> **거부되는 예제 (rejected) — 무더기를 쓰겠다면서 그 권한을 안 받는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_alloc_nocap .
>
> proc f output u8 . effects alloc .    rem `input k cap allocator .` 가 없다
> do
>   return 1 .
> end .
> ```
>
> 진단: `E-ALLOC-NOCAP`

7.2.2 들어오는 자리와 나가는 자리

1 프로그램이 시작하는 op 은 **권한만** 받는다(`E-ENTRY-PARAMS`). 자료를 받지 아니한다 — 건네줄 사람이 없기 때문이다.

2 시작 자리가 건네받을 수 있는 권한은 **닫힌 열**이다 — `args` · `env` · `io` · `allocator` · `heap` · `file_system` · `net` · `process` · `tty` · `clock` · `random` · `atomic`. (`process` — 다른 프로그램을 띄우고 지켜보는 권리.) 그 밖의 권한을 시작 자리가 요구하면 거부된다(`E-ENTRY-CAP`) — 아무도 줄 수 없는 권리를 적는 것은 장식이고, **장식은 거짓말이다.**

> **쉬운 말로**
>
> 왜 일곱뿐인가. 시작 자리의 권한은 **실행하는 쪽이 실제로 건네 줄 수 있는 것**이어야 한다. 목록에 없는 권한(예컨대 원자·기계 명령·장치)은 프로그램이 **스스로 지어낼 수 없고**, 그렇다고 실행하는 쪽이 건넬 수 있는 것도 아니다 — 그런 권한은 그것을 줄 자격이 있는 자리에서 만들어져 인자로 흘러야 한다. 장치(`mmio`)와 기계 명령 (`machine`)이 그렇고, 씨로 들어가는 문(`c`)도 그렇다.

3 다른 모듈의 이름은 그쪽이 `export` 한 것만 쓸 수 있다(`E-VISIBILITY`). 이름 앞에 모듈을 적는다고 없던 권한이 생기지 아니한다.

> **쉬운 말로**
>
> 이 절들이 되풀이하는 한 가지 — **권한은 건네받는 것이지 주워 쓰는 것이 아니다.** 무더기도, 씨로 들어가는 문도, 기계 명령도, 장치도, 시작 자리도 같다. 어디서 받았는지 적히지 않은 힘은 이 언어에 없다.

8 메모리와 소유 (Memory and ownership)

1 이 조항은 값이 **어디에 살고 언제 사라지는가**를 정한다. 로우엔트에는 쓰레기 수집기가 없고, 손으로 해제하다 생기는 잘못도 없다 — 그 둘 대신 번역 시점의 규칙이 있다.

8.1 값이 사는 곳

1 **저장되는 값**은 어느 영역 (region) 에 산다. 영역은 타입과 함께 정해지며 소스에 적혀 있다.

1a 모든 값이 영역을 갖는 것은 아니다. 계산 중에만 있는 작은 값은 저장될 자리를 갖지 않을 수 있으며, 그런 값에는 영역이 없다. 영역은 **값이 어디에 저장되는가**를 말하는 것이지 값이 존재하는 방식을 말하는 것이 아니다.

2 저장되는 값의 영역은 세 갈래다.

표 56 — 값이 사는 곳

갈래	설명
지역	op 안에서 나고, op 이 끝나면 사라진다. 가장 흔하다
정적	프로그램이 사는 동안 계속 있다
얻은 것	뿌리에서 얻는다. 누가 돌려줄지가 소유로 정해진다(§8.4)

2a 얻는 뿌리는 **둘**이다. 둘은 따로 깎이고 따로 되감긴다.

표 57 — 두 뿌리

뿌리	무엇으로 깎나	효과	성질
고정 창	`cap allocator · heap` 이 아닌 영역	`alloc`	**자라지 않는다.** 다 쓰면 `none` 이다. 운영체제가 없는 기계에서는 링커가 주는 두 경계 사이다 — 크기가 실행 파일에 박히지 않고, 기기가 부팅할 때 그 자리가 쓰인다
힙	`cap heap · region <이름> heap`	`heap`	**자란다.** 실행 중에 더 받아 오며, 이미 준 바이트를 옮기지 아니한다. 운영체제가 있는 기계에만 있다

3 값이 어느 영역에 사는지는 **소스에 적혀 있다.** 처리기가 몰래 옮기지 아니한다.

3a 뿌리에서 얻는 것 말고도, `some·ok·spawn actor` 가 만드는 값은 처리기의 **유한 풀**에 산다 (원소 나열 `lit array·lit slice` 의 자리와 수명은 §6.2.6 (7) 이, 구조체 값 `lit` 의 자리는 §6.2.6 (7c-1) 이 정한다 — 그 op 의 틀에 살고, `some·ok` 의 짐 · 그릇 · 통로 · 편지 · 태스크에 들어갈 때 값으로 베껴져 풀에 산다). 이 풀은 그 op 의 암묵 ⟦frame⟧ 이다 — 효과가 없고, `alloc·heap` 을 적지 않으며, 권한이 필요 없다. 풀은 루프를 한 바퀴 돌 때마다 되감기고, 동시에 살아 있는 값이 풀의 크기를 넘으면 그 자리에서 **멈춘다**(값이 아니라 중단이다). 크기는 기계가 정하며(운영체제가 없는 기계는 작다) 짓는 사람이 조절할 수 있다. 이 크기는 규범이 아니라 처리기가 밝히는 사실이다.

3a-1 처리기는 진입 op 마다 가장 깊은 부름 길이 풀에 남길 수 있는 값의 **상한**을 번역 때 센다 — 그 길에서 값을 만드는 자리의 수를 더한다(루프 안의 자리는 한 바퀴 몫이다 · 틀에 사는 구조체 값은 세지 않는다). 셀 수 없는 길(재귀 · 칸에 쌓기 · 되감을 수 없는 루프)은 수 대신 그 까닭을 밝힌다. 센 상한이 그 기계의 풀보다 크면 그 프로그램은 **적합하지 아니하다**(`E-POOL-BOUND`). 이 셈은 `--stack-report` 가 보인다(§6.2.6 (7c) 의 틀 셈과 같은 자리).

3b 뿌리의 커서는 실행 흐름 사이에 나뉘지 아니한다. 그러므로 태스크로 띄우는 op 은 뿌리에서 얻지 못한다 (`E-ALLOC-TASK`).

3c 얼로케이터 하나가 한 자리에서 **둘 이상의 흐름에 동시에 놓이면** 적합하지 아니하다 (`E-ALLOC-SHARED`) — 태스크 둘에 건네거나, 태스크에 건넨 채 곁에서 또 쓰는 것이다. 그 `reserve` 가 커서를 원자적

으로 옮기면(atomic 효과) 그렇지 아니하다. 태스크마다 **제 얼로케이터**를 주는 것은 적합하다 — 커서가 하나씩이므로 겨룰 것이 없다.

> **쉬운 말로**
>
> (3c) 는 2026-09-16 까지 「얼로케이터를 태스크에 건네는 것」 자체를 막았다. 그래서 진단문이 권하는 «태스크마다 제 얼로케이터를 줘라» 를 그대로 해도 같은 진단으로 거절됐다(결함 노트 75). **진단이 권하는 길이 막혀 있으면 그 진단은 길을 여는 것이 아니라 닫는 것이다.**

> **쉬운 말로**
>
> 흔히 「스택」과 「힙」이라 부르는 것이 각각 지역과 얻은 것에 해당한다. 이름을 달리 쓰는 이유는, 이 언어에서는 그것이 기계의 구조가 아니라 **값의 성질**이기 때문이다.

8.2 영역을 받는 자리

1 영역은 **인자로 건네받는다**. `input <이름> region <타입> . .` 으로 적으며, 뒤의 타입은 그 영역을 부르는 이름이다.

2 영역에서 자리를 얻는 op 은 `alloc` 효과를 갖는다(§7.1). 곧 **몰래 할당하는 op 이 없다** — 자리를 쓰는 op 은 계약에 그렇게 적혀 있다.

3 영역에서 얻은 것은 그 영역보다 오래 살 수 없다. 영역이 끝나면 거기서 얻은 것도 끝난다.

3a 영역은 낱낱이 돌려주지 아니한다. 영역이 끝날 때 **한꺼번에** 걷힌다.

> **예제 (example) — 영역을 받아 자리를 얻는다**
>
> **소스 코드 (source)**
>
> ```
> module ex_region .
>
> type scratch u64 .
>
> proc build input temp region scratch . output u64 . effects alloc .
> do
>   let s stack u64 stack_new temp capacity 4 . .
>   push s. 10 .
>   push s. 20 .
>   return 2 .
> end .
> ```

> **쉬운 말로**
>
> `effects alloc` 이 적혀 있는 것에 주의한다. 자리를 얻는 것은 **효과**이므로, 순수한 op(`fn` 에 `effects none`)은 영역을 쓸 수 없다. 어디서 메모리가 나오는지가 서명에 적혀 있다는 뜻이다.

4 영역을 여는 문은 이름과 **종류**를 함께 적는다: `region <이름> <종류> do … end .`. 종류는 닫힌 여덟이다 — `stack` · `frame` · `arena` · `static` · `heap` · `mmap` · `disk` · `device`. 그 밖의 낱말은 적합하지 아니하다(`E-REGION-KIND`).

4a 영역 안에서 얻은 값을 그 영역 **밖으로 들고 나가는** 것은 적합하지 아니하다 (`E-REGION-ESCAPE`). 영역이 닫히면 그 값은 없으므로, 밖에 남은 이름은 없는 것을 가리킨다. 들고 나가는 자리는 `return`, 영역 밖 이름에 대입하기, 그리고 영역 밖 이름의 **칸이나 원소에** 대입하기(`set field h. store . b. .`)다. 영역의 바이트를 들지 않는 값 — 정수·참거짓처럼 스칼라 타입으로 묶인 것, `len·idx` 처럼 스칼라를 내는 식 — 은 들고 나가지 아니한다. 영역의 바이트를 드는지는 **흐름을 따라** 가린다: op 부름의 결과는 그 op 의 몸에서 결과로 흘러드는 입력의 것만 든다. 결과가 `lit` 로 지은 묶음이면 칸마다 따로 가린다 — 영역의 바이트가 든 칸을 꺼내 들고 나가면 적합하지 아니하고, 들지 않은 칸은 상관없다. 몸이 결과를 지역에 담았다가 돌려주는 등 흐름을 가를 수 없으면 입력 모두에서 나온 것으로 본다.

거부되는 예제 (rejected) — 영역의 슬라이스를 바깥 묶음의 칸에 넣는다

소스 코드 (source)

```
module ex_region_field .

struct holder do store mut slice u8 . end .

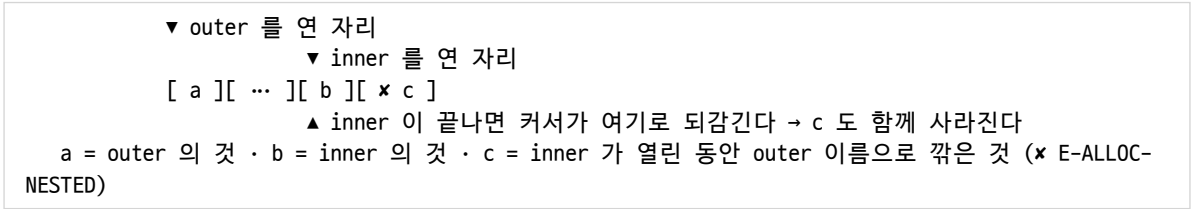
proc f output u64 . effects alloc . do
  var h lit holder do store subslice "abcd" 0 0 . . end . .
  region r arena do
    let g option mut slice u8 alloc_bytes r. capacity 16 . .
    if is_some g. . do
      let b mut slice u8 some_value g. . .
      set field h. store . b. .
    end .
  end .
  return len field h. store . . .
end .
```

진단: `E-REGION-ESCAPE`

4b 그 op 의 영역 매개변수가 아닌 이름으로 영역을 여는 것도 적합하지 아니하다 (`E-REGION-UNDEF`).

4c 같은 뿌리(§8.1 (2a))의 영역이 안쪽에 열려 있는 동안 **바깥 출처의 이름으로 깎는** 것은 적합하지 아니하다(`E-ALLOC-NESTED`). 바깥 영역 블록의 이름도, 영역 매개변수도, 같은 뿌리의 권한(`cap allocator` 는 고정 창 · `cap heap` 은 힙)도 그렇다. 뿌리마다 깎는 자리가 하나라서 안쪽 영역이 끝날 때 그 바이트까지 되감기기 때문이다. 다른 뿌리의 영역은 안쪽에 열려 있어도 상관없다.

도해 (diagram) — 뿌리마다 커서는 하나 — 안쪽 영역이 끝나면 커서가 되감긴다



거부되는 예제 (rejected) — 안쪽 영역이 열린 채 바깥 영역에서 깎는다

소스 코드 (source)

```
module ex_region_nested .

proc f output u64 . effects alloc . do
  region outer arena do
    region inner arena do
      let g option mut slice u8 alloc_bytes outer. capacity 8 . .
    end .
  end .
  return 0 .
end .
```

진단: `E-ALLOC-NESTED`

4d 영역 안에서 얻은 바이트를 **영역 밖에서 태어난 actor** 에게 보내는 것은 적합하지 아니하다 (`E-ALLOC-OUTLIVES`). actor 는 영역보다 오래 살고, 받은 슬라이스를 간직하는지는 번역할 때 보이지 않는다. actor 를 영역 안에서 만들거나, 영역보다 오래 사는 메모리를 건넨다.

4e 영역 블록을 나가는 **모든 길** — 끝에 닿기, `return`, 영역을 감싼 루프의 `break`·`continue` — 에서 그 영역은 되감긴다. 영역 블록의 이름을 인자로 건네면 받은 쪽은 그것을 영역 매개변수로 받아 같은 뿌리에서 깎는다.

5 종류는 **어느 뿌리에서 깎는가**를 정한다(§8.1 (2a)): `heap` 은 힙에서, 나머지 일곱은 고정 창에서 깎고 되감는다. 두 뿌리의 영역은 한 op 안에서 겹쳐 열 수 있으며, 안쪽 영역을 되감아도 다른 뿌리의 바깥 값은 남는다. 이 판의 처리기는 `heap` 이 아닌 일곱을 **서로 다르게 다루지는 아니한다** — 그 일곱은 **어디서 저장소가 오는지**를 읽는 사람에게 말하는 낱말이며, 처리기가 그것을 가려 쓰는 것은 뒷판의 일이다.

> **참고 (informative)**
>
> **어휘를 닫는 것과 동작을 가르는 것은 다른 일이다.** 아무 낱말이나 쓸 수 있으면 그 낱말은 아무것도 말해 주지 못한다 — 여덟으로 닫았기에 `region t arena` 를 읽은 사람이 "범프 할당이고 한꺼번에 되돌린다"를 안다. 동작의 구별은 **뿌리 둘**(고정 창 · 힙)까지 있고, 나머지 일곱 사이의 구별은 아직 없다. 그 사실을 여기 적어 둔다.

8.3 옮기기와 베끼기

1 값을 다른 이름에 넣을 때, 그 값이 소유를 가진 것이면 **옮겨진다**(원래 이름은 더는 쓸 수 없다). 소유가 없는 값이면 베껴진다.

2 옮겨진 뒤에 원래 이름을 쓰면 번역이 거부된다.

8.3.1 값이 놓이는 자리 — 임시값과 목적지

1 값을 만드는 식이 **놓일 자리**(목적지 (destination))를 가질 때가 있다. `let` 의 초기값, `return` 의 값, 인자 자리, 묶음의 필드가 그러하다.

2 다음 셋이 모두 성립하면 그 값은 **목적지에 곧바로 만들어진다** — 중간 임시값도, 그것을 옮기는 일도 일어나지 아니한다.

표 58 — 곧바로 만들어지는 세 조건

조건	뜻
목적지가 **비어 있다**	`let` 초기화 · 아직 안 채운 필드 · 옮겨져 빈 자리
값이 **도중에 실패하지 아니한다**	목적지에 닿기 전에 빠져나가는 길이 없다
목적지가 **그 값만의 것**이다	같은 자리를 두 값이 다투지 아니한다

3 조건이 안 맞으면 임시값 (temporary) 이 만들어진다. 임시값은 그것을 감싸는 가장 작은 범위에 놓이고, 그 범위가 끝날 때 **만든 차례의 거꾸로** 없어진다.

4 임시값을 가리키는 참조는 그 범위 밖으로 나갈 수 없다(§8.4.1).

5 값을 옮기는 별도의 연산은 없다. 옮기기는 **자리를 바꾸는 일**이지 값을 만들어 내는 일이 아니다.

> **참고 (informative)**
>
> 왜 «곧바로 만들어진다» 를 규범으로 적는가. "할 수 있으면 한다"로 두면 같은 소스가 처리기에 따라 복사를 하기도 하고 안 하기도 한다 — 그러면 **비용이 소스에서 안 보인다**. 위 세 조건은 읽는 사람이 소스만 보고 셀 수 있는 것들이라, 조건이 맞으면 복사가 **없다고 믿어도 된다**.

> **참고 (informative)**
>
> 씨 플러스 플러스(C++)는 이 자리를 «값 범주» 와 «옮김 생성자» 로 푼다. 로우엔트는 그 어휘를 들이지 아니한다 — 옮기기가 **자리의 일**이지 타입의 일이 아니기 때문이다. 그래서 옮김을 위한 특별한 선언도, 그것을 부르는 규칙도 없다.

8.4 참조

1 참조 (reference) 는 다른 곳에 있는 값을 가리킨다. 두 갈래가 있다.

> **ref** (shared reference)
>
> 읽기 참조. 여럿이 동시에 가질 수 있다. 이것으로는 값을 고칠 수 없다.

> **mut_ref** (exclusive reference)
>
> 쓰기 참조. 한 값에 대해 **한 번에 하나만** 있을 수 있으며, 그동안 읽기 참조도 있을 수 없다.

2 이 규칙을 어기면 번역이 거부된다.

3 참조는 값이 아니다. 참조를 받은 이름을 셈이나 비교에 그대로 쓰는 것은 적합하지 아니하다(E-TYPE-REFVAL) — 가리키는 값을 쓰려면 `deref` 로 **적어서** 꺼낸다.

4 구조체의 칸을 따로 빌리는 표기는 이 판에 없다. 칸을 빌리려 적는 것은 적합하지 아니하다(E-BORROW-FIELD) — 빌림은 값 전체를 단위로 한다(§8.12).

> **거부되는 예제 (rejected) — 참조를 값처럼 셈에 쓴다**
>
> **소스 코드 (source)**
>
> ```
> module ex_ref_value .
>
> fn twice input p ref u64 . output u64 .
> do
>   return add p. p. . .    rem 가리키는 값은 `deref p` 로 꺼낸다
> end .
> ```
>
> 진단: E-TYPE-REFVAL

8.4.1 참조는 자기가 가리키는 것보다 오래 살 수 없다

1 지역 값을 가리키는 참조가 그 값이 사라진 뒤에도 남는 것을 탈출 (escape) 이라 한다. 로우엔트는 탈출을 **번역 시점에 거부한다**.

2 곧 op 안에서 만든 값을 가리키는 참조를 그 op 밖으로 돌려줄 수 없다.

> **예제 (example) — 읽기 참조는 여럿이 함께 가질 수 있다**
>
> **소스 코드 (source)**
>
> ```
> module ex_ref .
>
> export fn sum_two input a ref u32 . input b ref u32 . output u32 .
>   requires le deref a. . 1000 . .
>   requires le deref b. . 1000 . .
> do
>   return add deref a. . deref b. . . .
> end .
> ```

> **거부되는 예제 (rejected) — 지역 값을 가리키는 참조를 돌려줄 수 없다**
>
> **소스 코드 (source)**
>
> ```
> module ex_escape .
>
> export fn leak output ref u32 .
> do
>   let here u32 42 .
> ```

```
  return ref here . .     rem `here` 는 이 op 이 끝나면 사라진다
end .
```

진단: `E-ESCAPE: reference to a local escapes the op (dangling)`

> **쉬운 말로**
>
> 위 코드가 만약 번역된다면, 돌려받은 참조는 **이미 사라진 값**을 가리킨다. 그 참조를 읽으면 무슨 값이 나올지 아무도 모른다. 그래서 이 언어는 그 코드를 아예 받아들이지 않는다 — 실행해 보고 아는 것이 아니라 **번역할 때** 안다.

> **주의 — 널 참조가 없다**
>
> 참조는 언제나 살아 있는 값을 가리킨다. "가리키는 것이 없음"을 나타내야 하면 `option` 을 쓴다(§6.2.8) — 그러면 꺼내기 전에 확인하도록 처리기가 강제한다.

> **참고 (informative)**
>
> 읽기 여럿과 쓰기 하나를 가르는 규칙은 두 가지를 한꺼번에 막는다. 하나는 **다른 흐름이 동시에 고치는 것**이고, 다른 하나는 **읽는 중에 값이 바뀌는 것**이다. 둘 다 소스를 읽어서는 알 수 없는 종류의 잘못이라, 규칙으로 없앤다.

8.5 소유와 없애기

1 소유 (ownership) 은 어떤 값을 없앨 책임이 누구에게 있는가를 말한다. `owned t` 는 소유를 가진 값이다.

2 소유를 가진 값은 **정확히 한 번** 없애져야 한다. 두 번 없애는 것도, 없애지 않고 버리는 것도 번역 시점에 거부된다.

> **도해 (diagram) — 소유는 옮겨 다닌다 — 없앨 책임은 언제나 한 자리에**

```
var h be owned buffer v .      h ──▶ [ 버퍼 ]            책임: 여기
consume h .                    h (빈 이름)  ──옮김──▶ consume 의 h ──▶ [ 버퍼 ]   책임: consume
drop h .   (consume 안에서)                              [ 버퍼 ] 없어짐        책임: 끝
h 를 다시 쓰면                  E-OWN-MOVED              (옮긴 뒤 사용)
```

3 값을 없애는 일은 **두 갈래**이며, 이 언어는 그 둘을 다르게 다룬다.

> **해제** (release)
>
> 메모리를 돌려주는 것처럼 **언제나 성공하고 기다리지 않는 일**. 실패가 없으므로 값의 수명이 끝나는 자리에서 **조용히 일어나도 된다**. 삼킬 실패가 없기 때문이다.

> **완결** (completion)
>
> 파일 닫기·버퍼 비우기처럼 **실패할 수 있고 기다릴 수 있는 일**. 조용히 일어나면 **그 실패를 건네줄 자리가 없다.** 그래서 반드시 저자가 적어서 불러야 한다.

4 어떤 값이 완결을 요구하는지는 **프로그램 자신의 선언이 정한다** — 그 타입을 값으로 받아 `result` 를 돌려주는 op 이 있으면, 그것은 "이걸 끝내는 일은 실패할 수 있다"는 선언이다.

5 완결이 필요한 값을 부르지 않고 버리면 적합하지 아니하다. 처리기는 진단을 낸다 (`E-OWN-INCOMPLETE`).

6 옮긴 값을 다시 쓰는 것도 적합하지 아니하다(`E-OWN-MOVED`).

7 완결이 필요한 값을 묶을 때는 **그 자리에 `owned` 라고 적어야** 한다 (`E-OWN-BARE`). 그 낱말이 없으면 아무것도 끝내기를 요구하지 아니하며, 타입이 말하는 것과 묶음이 말하는 것이 갈린다.

8 갈래가 나뉘었다가 다시 만나는 자리에서, 소유의 상태는 **모든 길에서 같아야** 한다 (E-OWN-JOIN). 한 길에서 없애고 다른 길에서 살려 두면, 만난 뒤의 그 값이 살아 있는지 아닌지를 **아무도 말할 수 없다.**

8a 되풀이를 떠나는 길도 길이다. break 로 되풀이를 떠나는 길은 그 되풀이의 뒤에서 여느 끝과 만난다 — 한쪽에서만 값을 옮겼으면 거부된다(E-OWN-JOIN). 수를 적은 break 는 그 수만큼 바깥 되풀이의 뒤에서 만난다. 조건이 true 인 while 은 break 로만 끝나므로 만나는 길이 없다 — 그 길의 상태가 곧 뒤의 상태다. 값을 옮긴 뒤 continue 로 다음 바퀴에 가는 것은, 다음 바퀴가 옮긴 값을 다시 만나므로 거부된다(E-OWN-MOVED).

9 묶음의 owned 칸 하나를 옮긴 뒤에 **묶음 전체를 다시 옮기는** 것은 적합하지 아니하다(E-OWN-PARTIAL). 받는 쪽은 온전한 묶음을 받았다고 여기는데 그 안의 한 칸은 이미 남의 것이다.

쉬운 말로

이 셋은 모두 같은 물음의 다른 얼굴이다 — "지금 이 값을 없앨 책임이 누구에게 있는가"에 **언제나 하나의 답**이 있어야 한다는 것. 답이 갈래마다 다르거나(8), 일부만 옮겨 갔거나(9), 애초에 묻지 않았다면(7) 그 값의 끝은 아무도 책임지지 않는 자리가 된다.

예제 (example) — 해제 — 실패할 수 없으므로 drop 이면 된다

소스 코드 (source)

```
module ex_own .

type buffer u8 .

fn sink input h owned buffer . output u8 .
do
  drop h. .
  return 0 .
end .
```

거부되는 예제 (rejected) — 두 번 없앨 수 없다

소스 코드 (source)

```
module ex_own_bad .

type buffer u8 .

fn twice input h owned buffer . output u8 .
do
  drop h. .
  drop h. .     rem 이미 없어진 것을 또 없앤다
  return 0 .
end .
```

진단: E-OWN-MOVED

쉬운 말로

두 갈래를 가르는 기준이 하나뿐이라는 점이 중요하다 — **끝내는 일이 실패할 수 있는가.** 실패할 수 없으면 언어가 알아서 치워도 아무 정보가 사라지지 않는다. 실패할 수 있으면 그 실패를 누군가 받아야 하고, 받을 사람은 저자뿐이다.

참고 (informative)

새 낱말을 만들지 않은 것도 눈여겨볼 자리다. 완결은 **보통의 op** 이다 — 소유한 값을 받고 `result` 를 돌려주는 함수. 그 모양 자체가 선언이 되므로 언어에 낱말을 더할 이유가 없었다.

쉬운 말로

「없애기를 잊었다」 를 번역 시점에 잡는다는 것이 이 조항의 핵심이다. 프로그램이 끝날 때 운영체제가 치워 주기를 기대하지 않으므로, 운영체제가 없는 환경에서도 같은 코드가 성립한다(§5.1).

스스로를 품는 마디 (self-owning node)

칸 하나가 `option (owned <자기 타입>)` 인 struct. 그 칸은 값이 아니라 얼로케이터 바이트에 지은 다른 마디의 **주소 한 칸**이라 크기가 정해지고, 마디끼리 이어 나무를 짓는다.

10 스스로를 품는 마디 (self-owning node) 는 칸 `<이름> option (owned <자기>) .` 를 가진 struct 다 (§6.2.7 (3b)). 마디는 얼로케이터가 준 바이트에만 지어진다 — 바인딩의 `use` 로 그 얼로케이터를 고른다: `let n use al. option (owned node) lit node do … end . .`. 자리가 모자라면 그 값은 `none` 이다. 마디는 **완결이 필요한 값**처럼 다룬다 — `owned node` · `option (owned node)` 를 옮기지도 돌려주지도 않고 버리거나(블록 끝 · 이른 `return`) 아직 든 값 위에 덮어쓰면 적합하지 아니하다(`E-OWN-INCOMPLETE`).

11 마디는 `drop <이름>. use <얼로케이터>. .` 로만 돌려준다. 그 얼로케이터가 낱낱이 돌려주는 것(`freeing_allocator`)이면 바이트가 그리로 돌아가고, 아니면 얼로케이터가 바이트를 든 채 둔다. 이름이 `option` 이고 비어 있으면 아무것도 하지 아니한다. 얼로케이터 없이 마디를 `drop` 하거나, 마디가 아닌 값에 `use` 를 적으면 적합하지 아니하다(`E-DROP-USE`).

12 자식 칸이 아직 든 마디를 돌려주면 적합하지 아니하다(`E-OWN-NODE-KIDS`) — 칸을 먼저 옮겨 낸다. 그 마디가 `option` 안에 있어 번역 때 보이지 않으면 실행이 그 자리에서 멈춘다(`E-VM-NODE-KIDS`). 깊은 나무는 되부름이 아니라 되풀이로 허문다.

13 `ref` 로 빌린 마디에서 자식을 꺼내 가면 적합하지 아니하고, `mut_ref` 로 빌린 마디에서 꺼낸 자식은 op 이 끝나기 전에 되돌려 놓아야 한다(`E-OWN-NODE-REF`) — 빌려준 쪽이 구멍 난 마디를 돌려받는다. 읽기만 하는 것(`is_some field t. left .`)은 꺼내는 것이 아니다. 자식을 꺼낸 마디를 통째로 옮기는 것은 (9) 와 같이 적합하지 아니하다(`E-OWN-PARTIAL`).

예제 (example) — 빌린 마디는 읽기만 한다

소스 코드 (source)

```
module ex_node_read .

struct node do
  key u64 .
  left option (owned node) .
end .

fn has_left input t ref node . output bool . do
  return is_some field t. left . . .
end .
```

거부되는 예제 (rejected) — 마디는 그 얼로케이터에 돌려준다

소스 코드 (source)

```
module ex_node_drop .

struct node do
```

```
  key u64 .
  left option (owned node) .
end .

fn lose input n owned node . output u64 . do
  drop n. .
  return 0 .
end .
```

진단: `E-DROP-USE`

쉬운 말로

마디가 얼로케이터를 들지 않는 까닭은 낭비다 — 얼로케이터는 나무 머리가 하나 든다. 그래서 마디는 «나를 준 곳» 을 모르고, 돌려줄 곳을 적는 사람은 저자다. 마디 타입을 모듈 밖에 내보내지 않으면 한 나무의 마디를 다른 얼로케이터에 돌려주는 실수는 그 모듈 안으로 좁혀진다(표준 라이브러리 `tree`).

8.6 메모리 안전이 어디까지 보장되는가

1 이 문서는 보장의 **강도를 갈라 적는다.** 그렇게 하지 않으면 지키지 못할 약속을 하게 된다.

표 59 — 보장의 강도

강도	뜻
증명됨	기계로 검사된 증명이 있다
정적	번역 시점의 규칙으로 막는다. 규칙 자체의 증명은 아직 없다
동적	실행 중에 검사한다. 어기면 트랩한다

2 참조의 배타 규칙, 탈출 금지, 소유의 한 번 없애기는 **정적**이다.

3 슬라이스 경계와 계약은 **동적**이되, 처리기가 증명하면 검사가 사라진다(§6.4.6).

주의 — 「안전하다」는 말은 무엇이 어떻게 안전한지 밝혀야 뜻이 있다

어떤 언어가 "메모리 안전"을 내세울 때, 그것이 번역 시점에 막는 것인지 실행 중에 검사하는 것인지, 증명된 것인지 규칙일 뿐인지가 갈린다. 이 문서가 강도를 갈라 적는 이유는 **못 지킬 약속을 하지 않기 위해서**다.

8.7 바이트를 다시 읽기 — `bit_cast` 와 `view`

1 같은 바이트를 **다른 타입으로 읽는** 일은 이 언어에서 두 갈래로 나뉜다. 갈래를 나누는 까닭은, 그 일을 통째로 「안전하지 않다」로 미루면 해악이 어디까지 번지는지 아무도 말할 수 없게 되기 때문이다.

표 60 — 바이트를 다시 읽는 두 갈래

갈래	op	무엇인가
값의 비트를 그대로 옮긴다	`bit_cast`	폭이 같은 두 스칼라 사이. 비트열은 그대로 두고 **읽는 법만** 바꾼다
바이트 위에 짜임을 얹는다	`view` · `try_view` · `view_array`	바이트 조각을 어떤 짜임의 배치로 **읽는다**. 베끼지 아니한다

8.7.1 `bit_cast` — 비트는 그대로, 읽는 법만

2 `bit_cast` 의 목표 타입은 **`plain` 인 스칼라**여야 한다. `plain` 이란 **모든 비트열이 쓸모 있는 값이고 덧댐이 없는** 타입을 말한다.

3 `bool` 과 열거 (enum) 은 `plain` 이 **아니다** — 쓸모없는 비트열(덫 표현 (trap representation))이 있기 때문이다. 그러므로 이 둘은 이 갈래 밖이며 거부된다 (`E-TYPE-BITCAST`).

3a 원천과 목표의 **폭이 같아야** 한다. 폭이 다르면 번역이 거부된다(`E-TYPE-BITCAST`) — 남거나 모자란 비트가 어디서 오는지 말할 수 없다. 폭을 먼저 `widen` · `narrow` 로 바꾼다.

> **주의 —**
>
> 「`plain` 인가」는 **직접** 물어야 한다. 폭 표에 없는 것을 `plain` 이 아니라고 에둘러 묻는 방식은 폭 표가 채워지는 날 조용히 뒤집힌다. 우연히 성립하던 판정은 그 우연이 사라질 때 아무 말 없이 사라진다.

> **거부되는 예제 (rejected) — `bool` 은 덫 표현이 있어 `plain` 이 아니다**
>
> **소스 코드 (source)**
>
> ```
> module ex_bitcast .
>
> fn f input a u8 . output bool .
> do
>   return bit_cast bool a. . .     rem 모든 비트열이 참·거짓인 것은 아니다
> end .
> ```
>
> 진단: `E-TYPE-BITCAST`

8.7.2 `view` — 배치를 얹는다

4 `view` 는 바이트 조각 위에 짜임의 배치를 얹는다. **베끼지 아니한다** — 그 자리의 바이트를 그대로 읽는다.

5 얹으려면 두 가지가 맞아야 한다.

표 61 — `view` 가 서기 위한 조건

조건	어기면	왜
조각이 배치만큼 길다	`E-VM-VIEW`	모자란 바이트를 읽는 것은 이 언어가 막는 바로 그것이다
시작 주소가 `align` 을 지킨다	`E-VM-ALIGN`	`align n` 은 **계약**이며, 계약을 검사하는 자리가 바로 이 경계다

6 `try_view` 는 같은 일을 하되, 서지 못하면 **거짓말 대신 없음**을 준다(⟦option⟧). 곧 실패가 값으로 돌아온다.

7 `view_array` 는 같은 짜임을 **여럿** 얹는다. 조건은 같다.

7a **줄 보기** — `view_array array m t b.` 는 줄 하나(`m` × ¦`t`¦ 바이트)를 원소 하나로 얹는다. 결과는 `slice array m t` 이고(§6.2 (9a) 의 줄의 나열과 같은 타입), `idx g. r.` 는 줄 `r` 을 보는 `slice t` 다. 조건은 (7) 과 같다 — `b` 의 길이가 줄 바이트의 배수가 아니면 멈추고(`E-VM-VIEW`), `t` 는 `plain` 이어야 하며(`bool` 은 `E-TYPE-BITCAST`), `m` 은 1 이상의 리터럴이다. 쓸 수 있는가는 `b` 를 따르고, 출처는 `b` 다(§8 (8a)) — 베끼지 않으므로 무효화 · 탈출의 규칙이 `b` 의 것 그대로 선다.

> **쉬운 말로**
>
> 정렬을 「빠르기 문제」로 여기면 어긴 자리가 조용히 지나가고, 어떤 기계에서만 어느 날 무너진다. 이 언어는 그것을 **계약**으로 부르고 경계에서 검사한다 — 짧은 조각을 거절하는 것과 **정확히 같은 이유**로 어긋난 주소를 거절한다. 둘 다 「없는 것을 있는 것처럼 읽는 일」이다.

8.8 고칠 수 있음은 적힌 대로만 흐른다

1 조각과 참조는 기본이 **읽기**다. 고치려면 그 자리에 `mut` 이 적혀 있어야 한다.

표 62 — 고칠 수 있음이 어긋나는 자리

진단	무엇이 어긋났는가
`E-TYPE-MUT`	`mut` 이라 적히지 않은 조각의 원소에 쓴다
`E-TYPE-REF`	나누어 가진 읽기 참조를 통해 쓴다 — 고치려면 `mut_ref` 여야 한다
`E-TYPE-ARGMUT`	읽기만 되는 값을 `mut·owned·mut_ref` 를 받는 자리에 건넨다
`E-TYPE-RETMUT`	고칠 수 있는 것을 내놓겠다고 적고 읽기만 되는 것을 돌려준다

2 곧 고칠 수 있음은 **한 방향으로만** 좁아진다. 고칠 수 있는 것을 읽기로 건네는 것은 되지만, 읽기만 되는 것을 고칠 수 있는 자리에 건네는 것은 되지 아니한다.

> **쉬운 말로**
>
> 이 규칙이 지키는 것은 **읽는 쪽의 믿음**이다. 어떤 값을 읽기로 받았다면 그것이 내가 보는 동안 바뀌지 않는다고 믿을 수 있어야 하고, 그 믿음은 아무도 몰래 쓰기 권한을 얻지 못할 때에만 선다. 그래서 좁아지는 방향은 하나뿐이다.

8.9 이름이 같아도 타입이 다르다

1 표현이 같아도 **이름이 다르면 다른 타입**이다(`E-TYPE-NOMINAL`). `newtype` 으로 가른 것들, 그리고 크기가 같은 서로 다른 정수 타입이 그렇다.

2 선언되지 않은 타입 이름은 거부된다(`E-TYPE-UNDEF`).

3 정수와 부동소수점 수는 **저절로 섞이지 아니한다**(`E-TYPE-MIX`). 건너려면 그렇게 적힌 op 을 쓴다.

4 타입이 자기 자신을 품는 것은 거부된다(`E-TYPE-CYCLE`). 처리기가 그 되풀이를 끝까지 따라가지 못하는 경우에도 **조용히 넘어가지 아니하고** 말한다(`E-TYPE-CYCLE-LIMIT`).

> **거부되는 예제 (rejected) — 정수와 부동은 저절로 섞이지 아니한다**
>
> **소스 코드 (source)**
>
> ```
> module ex_mix .
>
> fn f input a u32 . input b f64 . output f64 .
> do
>   return add a. b. . .     rem 건너려면 그렇게 적힌 op 을 쓴다
> end .
> ```
>
> 진단: `E-TYPE-MIX`

8.10 링커가 주는 저장소 — `storage reserved`

1 `storage reserved .` 블록은 **링커가 그 타입에 떼어 주는** 정적 저장소를 연다. 프로그램이 실행 중에 얻는 것이 아니라, 지어질 때 이미 자리가 잡혀 있는 것이다.

2 이 블록을 여는 op 은 다음 셋을 갖추어야 한다.

표 63 — `storage reserved` 가 갖추어야 하는 것

갖출 것	없으면	왜
권한	`E-RESERVE-NOCAP`	링커가 건네는 자리도 **건네받는 것**이지 주워 쓰는 것이 아니다

`effects state`	`E-RESERVE-NOEFFECT`	프로그램이 사는 동안 남는 자리이므로, 그것을 만지는 일은 상태를 만지는 일이다
기계가 그것을 줄 수 있음	`E-RESERVE-NOHOST`	운영체제 위에서는 링커가 그 자리를 그렇게 주지 아니한다

8.11 괄호는 블록을 넘지 아니한다

1 열린 `(` 는 **같은 블록 안에서** 닫혀야 한다. 닫히지 않은 채 블록이 끝나면 거부되고 (`E-PAREN-ESCAPE`), 아예 닫히지 않으면 거부된다(`E-PAREN-UNCLOSED`).

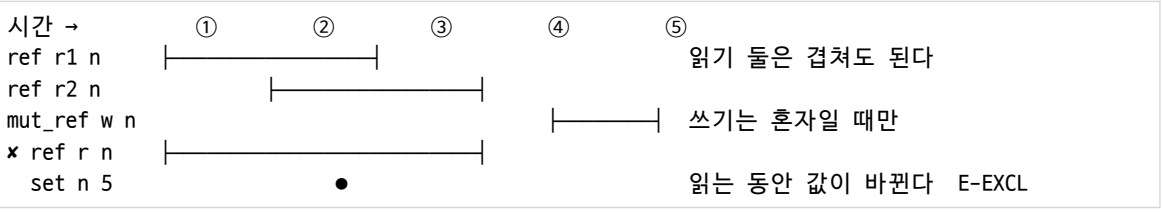
2 짝 없는 `)` 도, 블록의 경계를 넘어 짝을 찾는 `)` 도 거부된다(`E-PAREN-STRAY`).

> **쉬운 말로**
>
> 괄호가 블록을 넘어 짝을 찾을 수 있으면, 괄호 하나를 빠뜨린 소스가 **다른 뜻으로 온전히 읽히는** 일이 생긴다. 그때 처리기는 잘못을 말하는 대신 다른 프로그램을 짓는다. 경계를 못 넘게 하면 빠뜨림은 언제나 빠뜨림으로 드러난다.

> **거부되는 예제 (rejected) — 블록이 끝나도록 괄호가 안 닫혔다**
>
> **소스 코드 (source)**
>
> ```
> module ex_paren .
>
> fn f output u8 .
> do
>   return (add 1 2 .
> end .
> ```
>
> 진단: `E-PAREN-ESCAPE`

8.12 배타 — 읽는 이 여럿 또는 쓰는 이 하나

1 어떤 값에 닿는 길은 한 시점에 **읽기 여럿**이거나 **쓰기 하나**이며, 그 둘이 겹치는 것은 적합하지 아니하다(`E-EXCL`). 겹치면 읽는 쪽이 보는 값이 **언제 바뀌는지** 아무도 말할 수 없다.

> **도해 (diagram) — 한 값 n 에 대한 빌림을 시간 순으로**
>
> 
>

2 빌린 것은 빌려준 값보다 **오래 살 수 없다.** 빌려준 값이 옮겨졌는데 빌림이 남아 있으면 적합하지 아니하다(`E-EXCL-MOVED`).

3 빌림은 그것을 연 블록 **밖으로 나갈 수 없다**(`E-BORROW-ESCAPE`).

4 액터에게서 빌린 것을 들고 그 액터에게 다시 말을 거는 것은 적합하지 아니하다 (`E-BORROW-EXCL`) — 그 액터가 자기 상태를 고치는 동안 우리는 그 상태를 보고 있다.

5 `let` 으로 묶은 이름은 **고칠 수 없다**(`E-IMMUTABLE`). 고칠 이름은 `var` 로 묶는다.

6 한 부름이 같은 저장소를 `mut` **자리 둘 이상**에 넘기는 것은 적합하지 아니하다 (`E-EXCL`). 쓰는 이가 둘이므로 (1) 을 그대로 어긴다. 어느 쓰기가 남는지는 부른 자리가 아니라 **피호출자의 문장 차례**가 정하며, 부르는 쪽 글에는 그것을 알 길이 없다.

6a 같은 저장소인지는 이름이 아니라 **자리**로 가른다: 한 이름의 별칭, 같은 저장소에서 잘라 낸 겹치는 조각, 같은 값의 **같은 칸**, 그리고 값 **전체와 그 칸**은 같은 저장소다. 한 값의 **서로 다른 칸**(`field s. a .` 와 `field s. b .`)은 다른 저장소다 — 칸을 `mut` 자리에 넘기면 그 칸의 자리가 넘어가므로, 서로 다른 칸 둘은 쓰기 자리 둘에 함께 넘길 수 있다.

7 같은 저장소를 `mut` 자리 **하나**와 읽는 자리에 함께 넘기는 것(제자리 연산)은, 피호출자가 그 두 입력의 짝을 `inplace <쓰기 입력> <읽기 입력> .` 절(§6.4.1 (3a))로 **선언했고** 두 저장소가 **같은 구간**일 때에만 적합하다 (`E-EXCL-INPLACE`). 같은 구간이란 같은 저장소의 전체이거나, 같은 저장소에서 잘라 낸 같은 시작·같은 끝의 조각이다. 구간이 어긋나게 겹치거나 한쪽이 다른 쪽의 일부이면(**부분 겹침**) 선언이 있어도 적합하지 아니하다 — 원소마다 읽고 쓰는 몸은 같은 구간에서만 옳고, 부르는 쪽은 피호출자가 저장소의 어디에 쓰는지 알 수 없다.

7a `inplace` 절은 이름 **둘**을 적는다: 앞의 것은 그 op 의 `mut` 입력, 뒤의 것은 그 op 의 다른 입력이다. 짝 하나에 절 하나를 적는다. 모양이 어긋나면 `E-INPLACE-FORM` 이다. 그 선언이 **참인지** — 몸이 같은 구간에서 옳게 도는지 — 는 몸의 모양 둘 가운데 하나로 보여야 한다: ⓐ 읽기 입력을 읽는 마지막 문장까지 쓰기 입력에 쓰지 아니하거나, ⓑ 그 마지막 문장 안에서 쓰기는 `set idx <쓰기>. V. .` 하나의 첨자 V 로만, 읽기는 `idx <읽기>. V. .` 같은 첨자로만 하고, 반복 안이면 같은 바퀴에서 쓴 뒤에 읽지 아니한다(원소마다 제자리). 같은 짝을 `inplace` 로 밝힌 op 에 넘기는 것은 그 op 의 선언에 맡긴다. 어느 모양으로도 보이지 않으면 `E-INPLACE-UNPROVEN` 이다. 이 검사는 보수적이다 — 옳은 몸을 거절할 수는 있어도 틀린 몸을 들이지 아니하며, 거절된 몸은 ⓐ 나 ⓑ 로 다시 쓴다.

7b 두 구간이 겹치는지 번역 때 가를 수 없을 때(잘라 낸 끝이 상수가 아닐 때) 처리기는 (6)·(7) 을 **실행 중에** 묻는다. op 은 들어설 때 슬라이스·뷰 매개변수의 짝마다 — 적어도 한쪽이 `mut` 이면 — 두 조각이 한 바이트라도 겹치는지 보고, 겹치면 멈춘다 (`E-VM-EXCL`, 네이티브는 같은 문장의 panic). 쓰기·읽기의 짝을 그 op 이 `inplace` 로 허락했고 두 조각이 같은 구간이면 지난다. 빈 조각은 아무것과도 겹치지 아니한다. 겹침은 바이트 자리로 가르므로 원소 폭이 다른 두 창도 같은 바이트를 보면 겹친다. 번역 때 이미 가른 부름(상수 구간)은 (6)·(7) 의 진단으로 거절되고, 이 검사는 겹치지 않는 부름의 답을 바꾸지 아니한다. 이 검사는 매개변수인 조각만 본다 — 구조체의 칸 안에 든 조각은 견주지 아니한다. 처리기는 한 짝의 견줌을 op 의 입구에서 뺄 수 있다. 다만 번역 단위 안에서 그 op 에 이르는 길이 모두 부름 자리이고(함수 값 · 액터 · 태스크로 닿지 아니하고), 그 **모든** 자리가 두 조각이 겹칠 수 없음을 이미 보일 때에만 그렇다. 보이는 꼴은 넷이다: ⓐ 같은 저장소에서 잘라 낸 두 조각이 비었거나, 한 조각의 끝이 다른 조각의 시작을 넘지 않는다 — 끝은 상수이거나, 같은 실행 중 값 `v` 에 대해 `k + m·v`(`k · m` 은 상수, $m \geq 0$) 꼴로 읽히는 식이다(0 에서 `k` 앞까지와 `k` 에서 `n` 앞까지 · 0 에서 `n` 앞까지와 `n` 에서 `2n` 앞까지). 그 짝을 `inplace` 로 허락했으면 같은 구간이어도 된다. `view_array` 로 다른 원소로 본 조각은 같은 바이트다. 상수 끝의 조각(또는 그것을 `view_array` 로 본 것)을 다시 자른 조각은 끝을 몰라도 그 조각 **안에** 있다 — `subslice` 는 끝이 밖이면 실행이 멈춘다(`E-VM-BOUNDS`) — 그래서 감싸는 두 상수 구간이 앞뒤로 갈리면 안 겹친다, ⓑ 두 조각이 부르는 op 의 두 입력(또는 그 안의 조각)이고 부르는 op 이 그 짝을 제 입구에서 이미 겹치지 않는다고 견주었다 (`inplace` 로 견준 짝이면 둘을 그대로 넘기고 받는 쪽도 그 짝을 `inplace` 로 허락할 때만), ⓒ 한 조각이 문자열 리터럴(또는 그 조각)이거나, 담는 값이 언제나 문자열 리터럴인 이름이다 — 쓸 수 있는 자리는 읽기 전용 바이트를 가리키지 못한다(§6.2.6 (7b)), ⓓ 한 조각이 부르는 op 이 스스로 지은 나열(§6.2.6 (7) ⓐ 의 틀 안 칸)이나 그 op 이 `alloc_bytes` 로 얻은 바이트(그 `some_value`)의 조각이고, 다른 조각은 다른 선언의 나열 · 다른 `alloc_bytes` 의 답이거나 부르는 op 의 입력이다 — 선언마다, 얻을 때마다 제 바이트를 가지므로(뿌리는 준 바이트를 다시 주지 않고, 되감기는 영역이 끝날 때뿐이다 — §8.1 (2a) · §8.2 (4c)) 서로, 그리고 부르기 전부터 있던 바이트와 겹치지 않는다. 같은 답을 두 이름에 꺼내 담으면 같은 바이트다. 조각을 `let` 으로 묶은 이름은 그 `let` 의 값으로 본다 — 그 이름과 값에 든 이름이 op 안에서 한 번만 묶이고 참조로 빌려주지 않을 때만(입력이면 몸이 다시 쓰지 않을 때만). 번역 단위 밖에서 들어오는 입구(C 가 부르는 이름)는 뺀 짝을 그 입구에서 견준다. 그래서 겹칠 수 있는 부름은 어느 길로 와도 견주어지고 멈춘다 — 뺌은 답도 멈춤도 바꾸지 아니한다.

쉬운 말로

(7b) 는 2026-11-04 까지 «묻지 아니한다» 였다. 그래서 `wr2 (subslice s 0 k) (subslice s 0 k)` 는 검사를 지났고 뒤 쓰기가 앞 쓰기를 덮었다(외부 신고 REQ-0036). 세 갈래 — 같은 이름·같은 수의 끝을 번역 때 같은 구간으로 보기, 한계로 적고 허용하기, 실행 중에 견주기 — 가운데 소유자가 셋째를 골랐다(2026-11-04). 처음 견주자 코퍼스의 부름 셋이 멈췄다: 버퍼 전체와 그 안을 `view_array` 로 잘라 낸 창을 함께 쓰는 자리에 넘긴 둘(정적 검사는 그 별칭을 못 따라갔다)과, 같은 구간을 받는다고 밝힌 op 의 감싸개가 그 허락(`inplace`)을 옮겨 적지 않은 하나. 셋 다 저장소를 겹치지 않게 잘라 넘기거나 허락을 적도록 고쳤다.

쉬운 말로

(7) 은 2026-09-21 까지 «읽기를 다 마친 뒤에만 쓰는 op» 에 대한 **지은 이의 의무**였고 도구가 보지 않았다(RFC-0115 §8-15 ⓒ). 그래서 어긴 부름은 진단이 아니라 틀린 답으로 나타났다. 이제 허락은 **op 의 머리에 적히고**, 부르는 쪽 한 자리에서 물을 수 있다(RFC-0116 D2 B1, 2026-09-22 소유자 결정 ⓑ). 옛 규칙이 허락하던 부분 겹침은 새 규칙이 막는다 — 코퍼스의 그런 부름 일곱은 저장소를 겹치지 않게 잘라 넘기도록 고쳤다.

거부되는 예제 (rejected) — 선언하지 않은 op 에 같은 저장소를 쓰기·읽기로 함께 넘긴다

소스 코드 (source)

```
module ex_inplace_undeclared .

proc scale input o mut slice u64 . input a slice u64 . output u64 . effects none . do
  set idx o. 0 . mul idx a. 0 . 2 . .
  return 1 .
end .

proc f input w mut slice u64 . output u64 . effects none . do
  return scale w. w. . .
end .
```

진단: `E-EXCL-INPLACE`

예제 (example) — 같은 구간을 허락한 op 의 제자리 부름

소스 코드 (source)

```
module ex_inplace_ok .

proc scale input o mut slice u64 . input a slice u64 . output u64 . effects none . inplace o a .
do
  set idx o. 0 . mul idx a. 0 . 2 . .
  return 1 .
end .

proc f input w mut slice u64 . output u64 . effects none . do
  return scale subslice w. 0 4 . subslice w. 0 4 . . .
end .
```

거부되는 예제 (rejected) — 같은 구간이면 쓴 뒤에 읽는 몸에 inplace 를 적는다

소스 코드 (source)

```
module ex_inplace_unproven .

proc late_read input o mut slice u64 . input a slice u64 . output u64 . effects none . inplace o
```

```
a . do
  set idx o. 0 . 1 .
  return idx a. 0 . .
end .
```

진단: E-INPLACE-UNPROVEN

8 op 은 머리에 invalidates <입력> . 절(§6.4.1 (3a))을 적어, 그 입력의 저장소에서 **나온 뷰를 무효로 만든다**고 밝힐 수 있다 — 블록을 돌려주는 것, 자라며 다른 자리로 옮기는 것, 되감는 것이 그렇다. 이름 **하나**를 적으며 그 op 의 입력이어야 한다 (E-INVALIDATES-FORM).

8a 한 op 의 몸 안에서, 뷰(타입에 slice 가 든 지역·빌린 이름)는 제 초기식에 나온 저장소를 **출처**로 든다 — 뷰에서 뷰를 만들면 출처를 잇는다. invalidates 를 밝힌 op 에 그 저장소를 넘긴 뒤 그 뷰를 쓰는 것은 적합하지 아니하다(E-VIEW-INVALIDATED). 이름을 통째로 다시 대입하면 새 출처로 되살아난다. 갈래는 **어느 한 갈래에서라도** 무효가 되면 합친 뒤 무효이고, 반복은 다음 바퀴의 머리까지 무효가 이어진다. 핸들·수 같은 **값**은 뷰가 아니다.

8b 뷰의 출처는 **뭉뚱그린다** — 부름의 결과는 그 인자 모두에서 나온 것으로 본다. 그래서 안전한 쓰기를 거절할 수는 있어도 무효화된 뷰를 놓치지 아니한다. op 경계는 이렇게 넘는다:

- invalidates 를 적지 않은 op 이라도, 제 몸에서 어떤 입력을 invalidates 를 밝힌 op(이렇게 추론된 op 포함)에 넘기면 그 입력을

무효로 만드는 op 으로 본다 — 감싸는 op 을 거쳐도 무효화가 사라지지 아니한다.

- 뷰를 묶음의 칸에 담으면(set field h. f . <뷰> .) 그 칸이 뷰를 든다. 그 칸의 출처가 무효가 된 뒤 칸을 읽어 쓰는 것은

E-VIEW-INVALIDATED 이고, 칸을 다시 대입하면 되살아난다.

8c 부수효과가 없는 op(effects none) 을 **같은 인자**로 두 번 불러 얻은 mut 뷰 둘은 같은 저장소로 본다. borrow 밖에서 둘을 함께 살려 두고 쓰면 (6) 을 어긴 것이다(E-EXCL) — 풀의 같은 핸들로 쓰기 뷰 둘을 꺼내는 것이 그렇다. 인자가 이름이나 상수가 아니면 같은지 묻지 아니한다.

거부되는 예제 (rejected) — 자라며 옮기는 op 뒤에 옛 뷰를 쓴다

소스 코드 (source)

```
module ex_view_invalidated .

struct buf do
  data mut slice u8 .
  n u64 .
end .

proc grow input b mut buf . output u64 . effects none . invalidates b .
do
  set field b. n . add field b. n . 1 . .
  return 1 .
end .

proc view_of input b buf . output slice u8 . effects none .
do
  return field b. data . .
end .

proc f input b mut buf . output u64 . effects none .
do
  let v slice u8 view_of b. . .
```

```
  let r u64 grow b. . .
  return len v. . .       rem `grow` 가 `b` 의 뷰를 무효로 만들었다 — 뷰를 다시 받는다
end .
```

진단: `E-VIEW-INVALIDATED`

쉬운 말로

이 규칙이 지키는 것은 §8.8 과 같다 — **읽는 쪽의 믿음**이다. 다만 §8.8 이 「쓸 수 있는가」를 타입으로 가른다면, 여기는 「지금 이 자리에 누가 함께 있는가」를 가린다. 타입이 맞아도 함께 있으면 안 되는 자리가 있고, 그것을 보는 것이 배타 규칙이다.

주의 —

`mut ref slice` 는 쓰지 아니한다(`E-MREF-SLICE`) — `mut slice` 가 이미 그 원소를 고칠 수 있게 하므로, 앞의 것은 **뒤의 것이 주지 않는 것을 하나도 주지 못한다.** 같은 것을 말하는 두 번째 철자다.

거부되는 예제 (rejected) — `let` 으로 묶은 이름은 고칠 수 없다

소스 코드 (source)

```
module ex_immutable .

fn f output u8 .
do
  let a u8 1 .
  set a. 2 .            rem 고치려면 `var` 로 묶어야 한다
  return a. .
end .
```

진단: `E-IMMUTABLE`

8.13 얼로케이터 — 권한 · 정책 · 상태

얼로케이터 (allocator)

뿌리에서 받은 바이트나 빌린 바이트를 **작은 자리로 나눠 주는** 값. 이 언어에서는 트레이트 `byte_allocator` 를 갖춘 actor 다.

1 뿌리(§8.1 (2a)) 위에서 자리를 나눠 주는 것을 얼로케이터 (allocator) 라 한다. 얼로케이터는 언어의 특별한 장치가 아니라 **트레이트 `byte_allocator` 를 갖춘 actor** 다. 그 말에는 세 가지가 있다.

표 64 — 얼로케이터를 이루는 셋

층	언제 있나	무엇인가
권한	번역할 때만	뿌리에 닿아도 되는가 — `cap allocator` · `cap heap`(⟦§7.2⟧)
정책	번역할 때(타입)	어떤 규칙으로 깎는가 — `byte_allocator` 를 갖춘 **타입**
상태	실행 중	커서·받침 바이트 — 그 타입의 **값**

2 actor 의 상태에 권한 칸(`root cap heap .` · `root cap allocator .`)을 둘 수 있다. 그 칸으로 `alloc_bytes` 를 부르면 그 권한의 뿌리에서 깎는다. 권한은 실행 중 값이 아니므로 그 칸은 크기가 없다.

3 권한 칸을 가진 actor 는 **같은 종류의 권한이 보이는 op** — 그 권한을 입력으로 받았거나, 스스로 같은 종류의 칸을 가진 actor 의 op — 에서만 띄울 수 있다(`E-CAP-FORGE`). 권한 없는 곳에서 한 줄로 뿌리에 닿는 길을 지어낼 수 없게 하기 위해서다.

4 운영체제가 없는 기계에서는 `cap heap` 칸을 가진 actor 와 `heap` 효과를 적은 actor 의 op 이 거부된다 (`E-HEAP-NOHOST`) — 단, 그 단위 어디서도 띄우지 않는 actor 는 묻지 아니한다.

5 표준 라이브러리는 기본 얼로케이터 둘을 이 모양으로 낸다: `allocs.fixed_bytes`(고정 창) · `allocs.heap_bytes`(힙). 그래서 기본 얼로케이터와 사용자가 지은 얼로케이터가 **같은 자리**에 들어간다 — 받는 쪽(`input comptime a type . input al a . requires allocs.byte_allocator a. . .`)은 그 둘을 가리지 아니한다.

6 `heap` 효과는 «뿌리가 자란다» 만 뜻한다. 자리를 **낱낱이 돌려주는** 것은 정책의 일이며, 그런 얼로케이터는 트레이트 `freeing_allocator` 를 갖춘다 — `byte_allocator` 의 op 에 더해 `release` 가 조각을 받아 돌려받았는지를 참거짓으로 답한다. 그래서 운영체제가 없는 기계에서도 고정 창 위의 낱낱 반환 얼로케이터를 쓸 수 있다.

7 얼로케이터가 «내가 마지막에 준 그 조각인가» 를 물을 때는 **길이가 아니라 정체**로 묻는다: `grow` 와 `release` 는 조각 자체를 받고, 기본 연산 `same_slice a. b. .` 로 두 슬라이스가 같은 자리에서 시작하고 길이가 같은지 본다. 길이만으로 알아보면 같은 길이의 남의 조각을 늘리거나 돌려받는다. `same_slice` 는 주소를 밖에 내지 않는다 — 답은 참거짓이다.

> **거부되는 예제 (rejected) — 권한 없이 권한 칸을 가진 actor 를 띄운다**

소스 코드 (source)

```
module ex_cap_forge .

actor grower do
  state do
    root cap heap .
  end .
  proc take input n u64 . output u64 . effects heap . do
    let g option mut slice u8 alloc_bytes root. capacity n. . .
    if is_some g. . do return n. . end .
    return 0 .
  end .
end .

proc f output u64 . effects heap state . do
  var g grower spawn actor grower . . rem `input h cap heap .` 가 없다
  return send g. take 8 . .
end .
```

진단: `E-CAP-FORGE`

8.14 객체마다 얼로케이터를 고르기 — op 머리의 `use` 절과 바인딩의 `use` · `keep`

1 op 은 머리에 `use <이름> <타입> .` 절을 적어 **자기가 깎아 쓰는 얼로케이터**를 밝힐 수 있다. `<타입>` 은 `byte_allocator` 를 갖춘 타입이거나, 그런 경계(`requires allocs.byte_allocator a. . .`)를 가진 comptime 타입 매개변수다. 본문에서 `<이름>` 은 평범한 이름이다.

1a 한 op 은 `use` 절을 **하나만** 적는다(`E-USING-DUP`). 절은 이름과 타입 두 낱말이다. 바인딩의 할당기 절은 이름 바로 뒤에 `use <출처>`(블록 끝에 돌려준다) 또는 `keep <출처>`(돌려주지 아니한다)로 적는다 (§6.5.1 (3c)) — 모양이 어긋나면 `E-USING-FORM` 이다.

1b 받는 쪽과 고르는 쪽이 **한 낱말**을 쓴다 — op 머리의 `use <이름> <타입> .` 은 «이 op 이 여기서 깎는다» 이고, 바인딩의 `use <출처>.` 는 «이 객체는 이것으로» 다. 머리의 `use` 는 가져오기(§6.10)와 자리로 갈린다: 가져오기는 선언 밖에 서고, 이 절은 op 의 이름과 `do` 사이에 선다. 머리의 절에는 돌려주기와 들고 나가기의 구별이 없다 — `keep` 은 바인딩에만 선다. 옛 철자 `using <이름> <타입> .` 은 거부된다(`E-USING-OLD`).

2 `use` 절은 **입력이 아니다.** 부르는 쪽은 그 얼로케이터를 위치로 적지 않으며, `<타입>` 이 타입 매개변수이면 그 타입 인자도 적지 않는다 — 얼로케이터의 타입에서 온다.

3 부르는 쪽은 바인딩에서 얼로케이터를 고른다: `let <이름> use <출처> <타입> <식> .` 고른 출처는 그 초기식의 **머리 호출**에 들어간다. 머리 호출이 `use` 절을 갖지 않은 op 이면 적합하지 아니하다 (`E-ALLOC-USING-UNUSED`) — 자기 얼로케이터를 이미 든 객체는 부르는 쪽의 선택을 받지 아니한다. 초기식이 `lit` 이면 그 바이트를 고른 출처에서 받는다(§6.2.6 (7e)). 스스로를 품는 마디(§8.5 (10))는 `owned` 로 묶으므로 블록 끝에 돌려주지 아니한다 — 돌려줄 책임은 그 값이 지고 다니며 §8.5 (11) 의 `drop … use` 가 돌려준다.

4 고르지 않으면 다음 차례로 기본값이 정해진다. 처음 맞는 것을 쓴다.

표 65 — 기본값을 정하는 차례

차례	출처
1	바인딩의 `use <출처>` · `keep <출처>`
2	이 op 자신의 `use` 절 이름(타입이 맞으면)
3	이 op 의 입력·바인딩 가운데 타입이 맞는 **하나뿐인** 것
없음	`E-ALLOC-NOSOURCE` — 보이는 얼로케이터가 없다
둘 이상	`E-ALLOC-AMBIGUOUS` — 짐작하지 않는다. 바인딩에 적어라

5 기본값은 **op 의 경계를 넘지 아니한다.** 출처는 그 op 의 서명과 본문에서만 오고, 부른 쪽의 얼로케이터가 저절로 흘러들지 않는다. 필드도 출처로 세지 아니한다. 곧 **전역 얼로케이터는 없다.**

5a 할당기의 선택은 **나무 위에서** 풀린다 — 처리기가 arity 로 폼을 세운 뒤, 단형화 앞이다. 나무를 세우지 않는 대조 방식으로 검사하면 풀리지 않은 절이 남으며, 처리기는 그것을 조용히 입력 하나 모자란 op 으로 검사하지 않고 `E-USING-UNRESOLVED` 로 말한다.

6 `alloc_bytes capacity <n>` 처럼 뿌리 피연산자를 생략할 수 있는 것은 **영역 블록 안**뿐이며, 그때 가장 안쪽 영역이 그 자리에 든다. 권한(`cap allocator`·`cap heap`)은 생략하지 못하고 이름으로 적는다 — 이름 없이 뿌리에 닿는 길을 만들지 않기 위해서다. 뿌리를 가리키지 않는 `alloc_bytes` — 영역 밖에서 생략했거나, 권한 입력 · 영역 매개변수 · 열린 영역 블록 · 권한 칸 가운데 어느 것도 아닌 이름을 댄 것 — 은 적합하지 아니하다(`E-ALLOC-NOROOT`). `stack_new` 의 뿌리는 영역 매개변수이며 생략하지 못한다(같은 코드).

6a 크기는 `capacity` 표식 뒤에 적는다: `alloc_bytes <뿌리> capacity <n>` · `stack_new <영역> capacity <n>`. 표식이 없으면 적합하지 아니하다(`E-ALLOC-CAPACITY-MARK`) — 생략형과 이름을 댄 형을 가르는 것이 그 표식이다.

> **쉬운 말로**
>
> 얼로케이터는 값이다 — 커서와 받침 바이트는 실행 중에 어딘가 살아야 한다. op 머리의 `use` 는 그 값을 **숨기지 않고** 적는 자리를 옮긴 것이다: 받는 쪽은 서명에 «나는 이 얼로케이터에서 깎는다» 를 적고, 부르는 쪽은 객체를 만드는 줄에 «이것으로» 를 적는다. 둘 사이의 기계어는 인자 하나를 넘기는 것과 같다.

> **예제 (example) — 한 op 에서 두 얼로케이터를 번갈아 쓴다**
>
> **소스 코드 (source)**
>
> ```
> module ex_using .
>
> trait carver do
>   reserve input s self . input n u64 . output u64 . effects state .
> end .
>
> rem 정책 둘 — 하나는 요청만큼, 하나는 두 배씩 센다
> actor exact do
> ```

```
  satisfies carver .
  state do
    used u64 .
  end .
  proc reserve input n u64 . output u64 . effects state . do
    set used. add used. n. . .
    return used. .
  end .
end .

actor doubled do
  satisfies carver .
  state do
    used u64 .
  end .
  proc reserve input n u64 . output u64 . effects state . do
    set used. add used. mul n. 2 . . .
    return used. .
  end .
end .

rem 받는 쪽 — 부르는 쪽은 얼로케이터도, 그 타입도 적지 않는다
proc take input comptime a type . use al a . input n u64 . output u64 . effects state . requires
carver a. . . do
  return send al. reserve n. . .
end .

proc main output u8 . effects state . do
  var e exact spawn actor exact . .
  var d doubled spawn actor doubled . .
  let x use e. u64 take 3 . .      rem 3
  let y use d. u64 take 3 . .      rem 6
  return narrow u8 add x. y. . . .
end .
```

거부되는 예제 (rejected) — op 머리의 얼로케이터 절은 use 다 — (1b) 를 시험한다

소스 코드 (source)

```
module ex_using_head_old .

trait carver do
  reserve input s self . input n u64 . output u64 . effects state .
end .

proc take input comptime a type . using al a . input n u64 . output u64 . effects state .
requires carver a. . . do
  return send al. reserve n. . .
end .
```

진단: `E-USING-OLD`

거부되는 예제 (rejected) — 보이는 얼로케이터가 둘인데 고르지 않았다

소스 코드 (source)

```
module ex_using_ambiguous .
```

```
trait carver do
  reserve input s self . input n u64 . output u64 . effects state .
end .

actor exact do
  satisfies carver .
  state do
    used u64 .
  end .
  proc reserve input n u64 . output u64 . effects state . do
    set used. add used. n. . .
    return used. .
  end .
end .

proc take input comptime a type . use al a . input n u64 . output u64 . effects state . requires
carver a. . . do
  return send al. reserve n. . .
end .

proc main output u8 . effects state . do
  var e exact spawn actor exact . .
  var f exact spawn actor exact . .
  let x u64 take 3 . .              rem e 인가 f 인가 — 짐작하지 않는다
  return narrow u8 x. . .
end .
```

진단: `E-ALLOC-AMBIGUOUS`

9 라이브러리 (Library)

1 이 조항은 개별 라이브러리 함수의 목록을 정하지 아니한다. 정하는 것은 **라이브러리가 지켜야 할 규칙**이다. 그 규칙을 지키는 한, 어떤 라이브러리든 이 언어의 보장 안에 있다.

> **쉬운 말로**
>
> C 표준은 언어와 라이브러리를 한 문서에 담고 함수 하나하나를 정한다. 로우엔트가 그러지 않는 이유는, 라이브러리가 언어보다 훨씬 빨리 바뀌기 때문이다. 함수 목록을 여기 적으면 이 문서가 곧 낡는다. 대신 **규칙**을 적으면 낡지 않는다.

9.1 라이브러리도 같은 언어로 쓰인다

1 표준 라이브러리의 **이식 계층**은 이 문서가 정한 언어로 쓰이며, 특권을 갖지 아니한다. 그 부분은 저자가 직접 쓴 코드와 같은 규칙을 받는다.

1a 다만 라이브러리 전체가 한 층인 것은 아니다. 바깥 세상에 닿는 자리에는 좁은 잎(leaf)이 있고, 그것은 `unsafe` 나 `extern` 을 써서 **신뢰 경계**(§4.4)를 넘는다. 그 경계는 좁게 유지되며 어디인지 소스에 적혀 있다.

1b 이름을 주소로 바꾸는 일(DNS)은 **연결을 거는 일과 다른 잎**이다. 둘 다 `cap net` 을 받고 `io` 효과를 내지만, 가르는 까닭이 있다: 주소를 이미 가진 프로그램은 이름 해석에 **아예 닿지 아니하고**, 검사는 주소를 손으로 주어 **망 없이** 연결을 잴 수 있다. 그리고 이름 해석은 기계와 시점에 따라 다른 답을 낼 수 있는데, 그 흔들림이 어느 줄에 있는지가 소스에 보인다.

> **쉬운 말로**
>
> 편함으로 치면 연결 잎 하나가 이름도 받는 편이 낫다. 그러나 그러면 "이 줄이 망에 몇 번 닿는가"를 소스만 보고는 알 수 없게 된다 — 이 언어가 권한과 효과에서 되풀이하는 것과 같은 물음이고, 같은 답을 준다: **감춘 것을 이름으로 드러낸다.**

2 곧 라이브러리의 op 도 계약을 적고(§6.4), 효과를 선언하며(§7.1), 권한을 인자로 받는다(§7.2).

> **참고 (informative)**
>
> 라이브러리에 특권이 없다는 것은 검사 가능성의 문제다. 라이브러리만 쓸 수 있는 숨은 통로가 있으면, 그 통로를 지나는 순간 프로그램의 보장이 끊긴다.

9.2 라이브러리가 지켜야 할 것

1 가) **실패를 값으로 돌려준다.** 특별한 값(−1 따위)으로 실패를 나타내지 아니한다 (§6.2.8).

2 나) **없애는 일이 실패할 수 있으면 그것을 숨기지 아니한다**(§8.5).

3 다) **권한을 스스로 지어내지 아니한다.** 필요한 권한은 인자로 받는다.

4 라) **효과를 정직하게 선언한다.** 안쪽에서 부르는 것의 효과까지 포함한다.

5 마) **기준이 갈리는 수는 어느 기준인지 적는다.** 행·열·색인·쪽 번호처럼 0 에서 세는지 1 에서 세는지가 갈리는 값은, 그 op 의 `lowdoc` 이나 모듈 계약이 **밝혀야** 한다. 이 언어의 색인은 0 기준이므로 라이브러리도 **0 기준을 기본**으로 하며, 바깥 규약이 1 기준이면(터미널의 ANSI 자리 지정 따위) 그 변환은 **op 안에서** 한다 — 부르는 쪽이 바깥 규약을 알아야 하면 그것은 소스 밖의 지식이다(§1.3).

> **참고 (informative)**
>
> 보기: `term.goto <버퍼> <자리> <행> <열>` 의 행과 열은 **0 기준**이다(화면 왼쪽 위가 `0 0`). ANSI 가 1 기준이라 op 이 안에서 1 을 더해 내보낸다 — 그 더하기는 부르는 쪽의 일이 아니다.

> **예제 (example) — 라이브러리도 같은 규칙을 지킨다**
>
> **소스 코드 (source)**

```
module ex_lib .

enum parse_error do
  too_short .
end .

rem 실패를 값으로 돌려준다 — 특별한 수(-1 따위)를 쓰지 않는다.
export fn first_byte input data slice u8 . output result u8 parse_error .
  errors too_short lt len data. . 1 . .
do
  guard ge len data. . 1 . else return error too_short . . .
  return ok idx data. 0 . . .
end .
```

> **쉬운 말로**
>
> 이 넷은 새 규칙이 아니라 **이미 정한 규칙을 라이브러리도 지킨다**는 말이다. 라이브러리라고 예외를 두면, 그 라이브러리를 쓰는 순간 프로그램의 보장이 끊긴다.

9.3 무엇이 언어이고 무엇이 라이브러리인가

1 §6.3.3 의 기본 연산과 빌트인 연산은 **언어**다. 이름이 고정되어 있다. 기본 연산의 이름은 저자가 가릴 수 없고, 빌트인 연산은 `call_builtin` 뒤에서만 서므로 저자의 이름과 만나지 아니한다.

1a 빌트인 연산은 라이브러리를 짓는 재료다. 표준 라이브러리는 빌트인 연산을 감싼 op 을 내놓고, 응용 프로그램은 그 op 을 부른다 — 빌트인 연산을 직접 부르는 것은 시스템 라이브러리와 처리기를 짓는 자리의 일이다.

2 그 밖의 모든 것은 라이브러리이며, 모듈을 가져와 쓴다.

3 어떤 기능이 언어에 들어가려면 다음을 만족해야 한다 — 라이브러리로는 표현할 수 없거나, 표현하면 비용이 숨는 경우.

4 언어와 라이브러리 사이에는 한 층이 더 있다 — 리프 (leaf) 다.

> **리프** (leaf)
>
> 운영체제나 하드웨어에 닿는 가장 얇은 조각. 로우엔트로는 쓸 수 없으므로 처리기가 준다. 리프는 **얇아야** 한다 — 로우엔트로 쓸 수 있는 것은 리프가 아니라 라이브러리다.

5 어떤 조각이 리프 자격을 갖는지는 다음 물음이 가른다 — "이것을 로우엔트로 쓸 수 있는가." 쓸 수 있으면 라이브러리다. **편의는 리프 자격이 되지 아니한다.**

표 66 — 세 층

층	누가 만드나	자격
언어	이 문서	라이브러리로 표현할 수 없거나, 표현하면 비용이 숨는다
리프	처리기	운영체제·하드웨어에 닿는다. 로우엔트로 쓸 수 없다
라이브러리	누구나	나머지 전부. 로우엔트로 쓴다

도해 (diagram) — 세 층 — 파일 하나를 읽을 때

```
library   files.open, files.read, files.slurp   Lowent code: owned handle, result
   ¦
leaf      file_open, file_read, file_close      from the processor, needs cap file_system
   ¦
OS        open, read, close                     the kernel
language  add, len, idx, narrow ...           fixed words (6.3.3), used at every layer
```

쉬운 말로

예를 들어 「파일을 열어 정수 핸들을 받는 것」은 커널에 닿으므로 리프다. 그 핸들을 **잊을 수 없는 값**으로 감싸는 것은 로우엔트로 쓸 수 있으므로 라이브러리다. 「파일을 통째로 읽어 주는 편의 함수」도 라이브러리다 — 편해서 리프가 되지는 않는다.

주의 — 편의는 언어에 들어가는 이유가 되지 아니한다

"이렇게 쓰면 편하다"는 라이브러리의 일이다. 언어에 낱말을 더하면 그 낱말은 모든 프로그램이 외워야 하는 것이 된다(§6.1.2). 그래서 언어는 **표현할 수 없는 것**만 받아들인다.

9.4 얼마나 익었는가 — 성숙도의 사다리

1 라이브러리 모듈은 저마다 **성숙도**를 적는다. 칸은 넷이며, 위 칸은 아래 칸의 의무를 **모두 포함한다**.

표 67 — 성숙도의 칸과 그 의무

칸	그 칸이 지는 의무
`experimental`	성숙도가 **적혀 있다**. 그 이상은 없다 — 그래서 실험이다
`incubating`	+ 주인이 되는 결정 문서가 **실제로 있다**
`standard`	+ 사람이 읽는 설명이 있고, 회귀 증인이 그 모듈을 **이름으로** 부른다
`deprecated`	+ 무엇으로 갈아타야 하는지 적혀 있다

2 적은 칸의 의무를 지키지 못하는 모듈은 적합하지 아니하다. 칸을 **올리는 것**은 의무를 갖춘 뒤의 일이며, 갖추지 못했으면 낮은 칸에 그대로 두어야 한다.

참고 (informative)

사다리의 값어치는 위 칸이 아니라 아래 칸에 있다. `experimental` 이라 적는 것은 "아직 믿지 말라"를 **소스에** 적는 일이고, 그 한 줄이 쓰는 사람에게 "이것에 기대어 설계해도 되는가"를 답해 준다. 칸이 없으면 모든 모듈이 똑같아 보이고, 그러면 **가장 익지 않은 것이 가장 익은 것처럼** 읽힌다.

참고 (informative)

칸은 **약속의 크기**이지 품질의 등수가 아니다. 잘 만든 실험 모듈이 허술한 표준 모듈보다 나을 수 있다 — 다른 것은 "바뀌지 않는다고 얼마나 말했는가"다.

9.5 무엇이 표준 라이브러리에 들어오는가

1 라이브러리에 들어오는 것은 **있어서**가 아니라 **증거로** 들어온다. 다음 여섯이 들어옴을 가르는 잣대다.

표 68 — 라이브러리 헌장

잣대	무엇을 뜻하나
이 언어로 쓸 수 있으면 이 언어로 쓴다	처리기 안의 잎(leaf)이 되는 것은 **바깥 권위를 쓰거나** 이 언어가 표현하지 못하는 일일 때뿐이다. **편함은 근거가 아니다**
시그니처가 비용과 권한의 장부다	효과·권한·부르는 쪽의 저장·할당·소유·완료·기계 제약이 **경계에서 보여야** 한다
셈과 권위를 가른다	꼴 짓기와 내보내기, 해석과 파일 열기, 난수의 다음 걸음과 엔트로피 얻기, 규약 부호화와 실어 나르기는 **각각 다른 것**이다
자리는 여러 축으로 가른다	권위·저장·실행 등급(⟦§10.8⟧)·프로파일(⟦§10.7⟧)은 **하나의 사다리로 눌러 담지 아니한다**
안에 있는 것은 작게 둔다	이식되는 바탕과 얇게 감사된 host 어댑터까지. 큰 바깥 의존·정책이 무거운 설비·아직 연구인 표면은 들이지 아니한다
모듈마다 증거가 있다	지지 경계를 선언하고, 단위·음성·오라클·실패·프로파일 관문을 갖춘다

2 이 문서는 라이브러리의 **목록을 적지 아니한다.** 무엇이 있는지는 소스에서 매번 다시 세어 내는 원장이 권위이며, 이 문서가 그것을 베끼면 **베낀 순간부터 갈라진다.**

> **쉬운 말로**
>
> 헌장이 목록을 겸하면, 아직 제안인 것이 표에 실렸다는 이유만으로 **이미 있는 것처럼** 보이게 된다. 실제로 그런 일이 있었다 — 있지도 않은 API 셋이 표에 있다는 이유로 사실로 읽혔다. 그래서 이 절은 **무엇이 들어오는가**만 정하고, **무엇이 들어와 있는가**는 세는 쪽에 맡긴다.

10 동시성 (Concurrency)

1 이 조항은 여러 일이 **함께 진행되는** 자리를 정한다. 동시성은 이 언어에서 특별한 기능이 아니라, §7 이 정한 효과 규칙이 그대로 적용되는 또 하나의 자리다.

10.1 동시성도 효과다

1 다른 실행 흐름을 만들거나 그것의 진행을 기다리는 일은 효과다. `wait` 와 `concurrent` 가 그것이며 (§7.1), 계약에 적어야 한다.

2 `wait` 와 `concurrent` 를 가르는 기준은 **누가 깨워 주는가**이다.

표 69 — 기다림의 두 갈래

효과	뜻
`wait`	중단될 수 있다. 깨우는 것이 **같은 프로그램의 다른 태스크가 아니다**(커널·장치가 깨운다)
`concurrent`	**동료가 돌아야** 끝난다. 그것을 돌려 줄 실행기가 있어야 한다

> **쉬운 말로**
>
> 예를 들어 파일에서 자료가 오기를 기다리는 것은 `wait` 다 — 깨우는 것이 같은 프로그램의 다른 태스크가 아니라 바깥이기 때문이다.

> **참고 (informative)**
>
> 이 구별이 왜 중요한가: 운영체제가 없는 작은 장치에는 「동료를 돌려 주는 실행기」가 없을 수 있다. 그때 `wait` 만 쓰는 프로그램은 그대로 돌지만 `concurrent` 를 쓰는 프로그램은 못 돈다. 효과에 적혀 있으므로 **번역 시점에** 알 수 있다.

10.2 액터

1 액터 (actor) 는 자기 상태를 가진 실행 단위다. 상태는 액터 안에만 있으며, 바깥에서 직접 건드릴 수 없다.

1a 그러므로 액터 값의 상태 칸을 **밖에서 읽는 것**은 거부된다(`E-ACTOR-FIELD`). 상태에 닿는 길은 메시지뿐이다 — 필요한 것을 돌려주는 op 을 액터에 둔다.

1b 상태 칸은 **빌림**(`ref·mut_ref`)일 수 없다(`E-ACTOR-STATE-REF`). 빌림은 빌려준 것보다 오래 살 수 없는데(§8.4.1) 상태 칸은 액터가 사는 동안 남고, 무엇을 빌렸는지 적을 자리가 없다.

1c `spawn actor` 로 갓 만든 액터의 상태 칸은 **모두 0** 이다. 0 은 슬라이스가 아니므로, 슬라이스나 빌림을 담는 칸을 **세우기 전에 읽는** 것은 적합하지 아니하다 (`E-ACTOR-UNINIT`). 처리기는 그 액터에 **처음 보내는 말**이 그런 칸을 세우지 않으면서 읽는 경우를 거부한다 — 곧은 문장 차례 안에서 본다.

> **쉬운 말로**
>
> (1c) 가 없던 2026-09-16 까지 그 자리는 **두 뒤끝이 갈리는 자리**였다: VM 은 「슬라이스가 아니다」로 멈추고 네이티브는 조용히 `none` 을 냈다(결함 노트 53). 둘 중 하나는 거짓말인데 어느 쪽인지 정본이 말하지 않았다. 답을 고르는 대신 **그 자리에 닿지 못하게** 한다.

1d **액터를 담는 이름은 리터럴 `0` 으로 시작할 수 있다** — 아직 띄우지 않은 액터다(§10.3 (2d) 의 핸들 0 과 같다). 그 밖의 수는 액터 자리에 들어가지 아니한다(`E-TYPE-LET` · `E-TYPE-VAR`). 0 인 액터에 `send` · `spawn send` · `try spawn send` 로 말을 보내거나 `drain` 하면 프로그램이 멈춘다(`E-VM-HANDLE-ZERO`). `spawn actor <이름>` 을 담은 뒤에 쓴다.

거부되는 예제 (rejected) — 0 이 아닌 수로 액터 자리를 채운다

소스 코드 (source)

```
module ex_actor_number .

actor counter do
  state do
    n u64 .
  end .
  fn get output u64 . do
    return n. .
  end .
end .

export proc f output u64 . effects state .
do
  var c counter 7 .
  return 0 .
end .
```

진단: `E-TYPE-VAR`

2 액터 안의 op 도 §6.4.1 의 갈래를 따른다 — 상태를 고치면 `proc` 이고 `effects state` 를 적어야 하며, 읽기만 하면 `fn` 이다.

3 상태를 고치면서 순수하다고 선언하면 번역이 거부된다.

4 액터의 op 이 `panic`(§6.5.9) 하면 그 액터는 **다시 세워질 수 있다.** 그때 그 액터의 상태는 처음으로 돌아가고 그 메시지가 다시 처리된다. 되살아나는 것은 **자기 상태뿐**이며, 다른 액터는 건드리지 아니한다.

4a 다시 세우는 것은 그 메시지가 **어떻게 왔는지와 상관없다** — `send` 로 기다리며 보낸 것이든, `spawn send` 로 우편함에 넣었다가 `drain` · `schedule`(§10.5)이 배달한 것이든 같다. 그 액터 앞으로 우편함에 **남아 있는 메시지는 건드리지 아니한다** — 다시 세운 액터가 그것을 넣은 차례대로 처리한다.

쉬운 말로

남은 메시지를 버리지 않는 까닭은 `spawn send` 가 이미 **받아 주었다**고 돌아왔기 때문이다. 그 뒤에 말없이 버리면 실패가 숨는다. 터진 메시지까지 다시 처리하는 것(4)과도 맞는다 — 잘못한 메시지는 다시 하면서 아무 잘못 없는 뒤의 메시지를 버릴 까닭이 없다. 같은 메시지가 계속 터지는 것은 (5) 의 정책이 끝을 정한다. 옛 상태를 보고 보낸 메시지가 처음으로 돌아간 상태에서 뜻이 없다면, 그 액터는 `failure restart never` 를 적는다.

5 언제까지 다시 세울지는 `failure restart <정책> .` 절이 정한다. 정책은 다음 셋이 전부다.

표 70 — 액터의 실패 정책 — 닫힌 집합 셋

정책	뜻
`max <수> [within <수> <단위>]`	그 수만큼 다시 세운다. 다 쓰면 위로 넘긴다
`never`	다시 세우지 아니한다 — 처음 터질 때 바로 위로 넘긴다
`always`	셈하지 아니하고 계속 다시 세운다

5b `within` 의 단위는 닫힌 셋이며, 하나와 여럿을 둘 다 받는다 — `second`/`seconds` · `minute`/`minutes` · `hour`/`hours`. 그 밖의 낱말은 적합하지 아니하다.

5c 처리기가 `within` 의 기간을 세지 못하면 그 사실을 **말한다**(`W-NOT-YET`). 기간을 적은 절을 조용히 `max <수>` 로 읽지 아니한다.

> **참고 (informative)**
>
> 하나와 여럿을 둘 다 받는 까닭은 읽기 위해서다 — `within 1 second` 와 `within 5 seconds` 는 사람이 쓰는 대로 적힌다. 이것은 **같은 것을 말하는 두 번째 방법**(§7.1.1)이 아니다: 뜻이 하나이고 철자가 수에 따라 갈릴 뿐이며, 처리기는 둘을 같은 수로 읽는다.

5a 실패 정책은 한 액터에 **하나**다(E-ACTOR-FAILURE). 편지함 정책도 그렇다 (E-ACTOR-MAILBOX, §10.5). 정책이 둘이면 어느 것이 서는지를 **아무도 말할 수 없고**, 그 답은 처리기의 속사정이 된다.

6 다 쓰고도 터지면 그 실패는 **위로 넘어간다.** 조용히 멈추어 있는 액터를 남기지 아니한다.

7 다시 세우는 것은 `panic` 에만 해당한다. 계약이 깨진 것(§6.4)은 다시 세우지 아니한다 — 그것은 프로그램이 스스로 적은 약속을 어긴 것이므로 다시 해도 같다.

> **쉬운 말로**
>
> 왜 「터지게 두는가」. 다시 세우는 쪽이 나은 실패가 있다 — 상태가 어쩌다 이상해진 경우다. 그런 실패를 하나하나 손으로 되돌리는 코드는 길고, 길면 그 코드 자체가 틀린다. 상태를 처음으로 돌리는 것은 **짧고 언제나 맞는 복구**다.

> **예제 (example) — 액터**
>
> **소스 코드 (source)**

```
module ex_actor .

actor counter do
  state do
    value u64 .
  end .

  rem 상태를 고치므로 `proc` 이고 `effects state` 다.
  proc inc output u64 . effects state .
  do
    set value. expr value. + 1 . .
    return value. .
  end .

  rem 읽기만 하므로 `fn` 이고 효과가 없다.
  fn get output u64 .
  do
    return value. .
  end .
end .
```

> **거부되는 예제 (rejected) — 상태를 고치면서 순수하다고 선언**
>
> **소스 코드 (source)**

```
module ex_actor_bad .

actor counter do
  state do
    value u64 .
  end .

  fn inc output u64 .
  do
    set value. expr value. + 1 . . rem 상태를 고치는데 `fn` 이다
```

```
    return value. .
  end .
end .
```

진단: E-EFFECT-PURITY

10.3 실행 흐름 만들기

1 spawn 은 새 실행 흐름을 만든다. 만드는 일은 효과이며 계약에 적어야 한다.

2 만든 흐름이 **끝나기를 기다리는지**, 아니면 떼어 놓는지가 효과로 갈린다 — 떼어 놓으면 detach 다.

2a spawn <op> [인자…] 는 그 흐름을 가리키는 **핸들**을 낸다. 핸들의 타입은 task T 이고 T 는 그 op 의 출력 타입이다. 핸들을 받지 아니하고 문장으로만 쓰면 그 흐름은 떼어 놓은 것이다.

2b await <핸들> 은 그 흐름이 끝날 때까지 기다렸다가 결과를 낸다. 결과의 타입은 핸들의 T 다. task T 가 아닌 값을 기다리는 것은 거부된다(E-TYPE-KIND). 기다리는 동안 다른 흐름이 나아간다 — 기다림이 기계를 멈추지 아니한다.

2c 끝날 수 없는 것을 기다리면 프로그램이 멈춘다. 처리기는 그 자리를 말하여야 한다.

2d **핸들 0 은 아무 흐름도 가리키지 않는 핸들이다.** task T 로 선언한 이름은 리터럴 0 으로 시작할 수 있다 — 흐름을 받기 전의 자리를 비워 두는 철자다. 그 밖의 수는 핸들이 될 수 없다(E-TYPE-LET · E-TYPE-VAR — 수는 흐름을 가리키지 아니한다). 핸들 0 을 기다리면 프로그램이 멈춘다(E-VM-HANDLE-ZERO). 처리기는 spawn 이 내는 핸들을 0 이 아닌 값으로 짓고, 핸들을 쓰는 모든 자리에서 0 을 거른다 — 비워 둔 핸들이 **다른 흐름을 조용히 가리키는 일은 없다.**

거부되는 예제 (rejected) — 0 이 아닌 수로 핸들을 짓는다

소스 코드 (source)

```
module ex_handle_number .

fn one output u64 . do
  return 1 .
end .

export proc pick output u64 . effects state concurrent .
do
  var h task u64 7 .
  task_group do
    set h. spawn one . .
  end .
  return await h. . .
end .
```

진단: E-TYPE-VAR

참고 (informative)

핸들 0 은 다른 언어의 널 포인터와 같은 자리다. 비워 둘 수 있어야 쓰기 편하지만, 비워 둔 것을 쓰면 엉뚱한 것을 집지 않고 **그 자리에서 멈춰야** 한다. 이 언어는 핸들을 1 부터 세어 0 이 어느 흐름과도 겹치지 않게 하고, 쓰는 자리마다 0 을 확인한다.

참고 (informative)

기다림이 효과인 까닭. await 는 값을 셈하지 아니하고 **시간을 쓴다** — 부르는 쪽은 그 값을 예산에 넣어야 한다. 그래서 순수한 op 은 기다릴 수 없다(§7.3).

3 `spawn actor` 는 액터를 만들고, `send` 는 그 액터에 메시지를 보낸다. 액터는 메시지를 **한 번에 하나씩** 처리한다.

3a 메시지에 값을 실어 보낼 수 있다 — `send <액터> <메시지> <값>…` 처럼 메시지 이름 뒤에 **나란히** 적는다. 받는 쪽에서 액터 인스턴스는 **첫 매개변수처럼** 다루어지며, 적은 값들이 그 뒤를 잇는다.

3b 소유를 가진 값을 보내면 그 소유가 **넘어간다.** 보낸 쪽은 그 값을 더 쓸 수 없으며, 쓰면 번역이 거부된다(§8.5).

> **쉬운 말로**
>
> 왜 넘기는가. 두 흐름이 같은 값을 함께 들고 있으면 경합이 생긴다. 소유가 메시지로만 옮겨 다니면 어느 순간에도 그 값을 만질 수 있는 흐름은 하나뿐이고, 그러면 경합은 **일어날 수 없다** — 막는 것이 아니라 있을 자리가 없다.

> **예제 (example) — 액터를 만들고 메시지를 보낸다**
>
> **소스 코드 (source)**
>
> ```
> module ex_spawn .
>
> actor counter do
>   state do
>     value u64 .
>   end .
>
>   proc inc output u64 . effects state .
>   do
>     set value. expr value. + 1 . .
>     return value. .
>   end .
> end .
>
> rem 액터를 만들고(`spawn`) 메시지를 보낸다(`send`).
> export proc use_counter output u64 . effects state .
> do
>   var c counter spawn actor counter . .
>   return send c. inc . .
> end .
> ```

> **쉬운 말로**
>
> 액터가 메시지를 한 번에 하나씩 처리한다는 것이 중요하다 — 그 안의 상태는 **동시에 건드려지지 않는다**. 자물쇠를 손으로 걸 필요가 없는 이유가 그것이다.

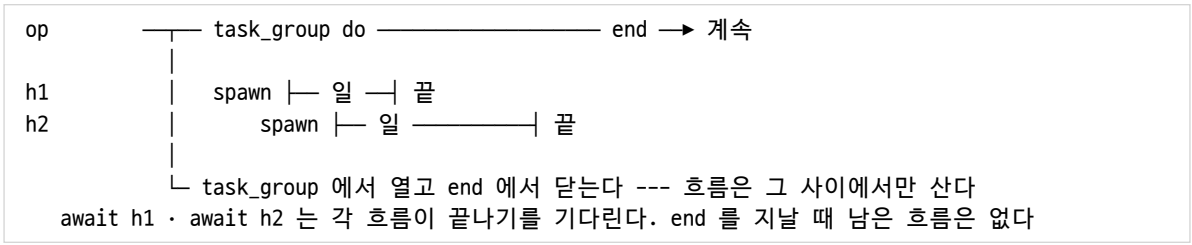
> **주의 — 동시성은 검사 밖의 세계가 아니다**
>
> 많은 언어에서 동시성은 "여기서부터는 조심하세요"의 영역이다. 로우엔트에서는 **같은 효과 규칙이 그대로 적용된다** — 무엇을 기다리는지, 상태를 고치는지, 실행기가 필요한지가 전부 계약에 적혀 있고 처리기가 검사한다.

10.4 흐름을 묶는 자리 — `task_group`

1 `task_group do <문장들> end .` 는 그 안에서 만든 흐름들을 **하나로 묶는다.** 블록이 끝나는 자리에서 묶인 흐름이 모두 끝나며, 하나라도 남은 채로 나가지 아니한다.

2 그러므로 흐름은 **자기를 만든 블록보다 오래 살지 못한다.** 이것이 이 언어가 흐름의 수명을 다루는 방식이며, 떼어 놓는 것(§10.3 (2))만이 그 규율의 예외다.

도해 (diagram) — 흐름의 수명은 task_group 블록 안에 갇힌다



3 그룹 안에서 `spawn <op> [인자…]` 는 **일감을 큐에 넣는다.** 같은 낱말이지만 `spawn actor <이름>` 은 액터 하나를 만드는 것이며, 둘은 뒤따르는 것으로 갈린다.

4 `task_group cancel_on_error do … end` . 는 **취소를 전파한다.** 묶인 흐름 하나가 오류로 끝나면, 아직 시작하지 아니하였거나 기다리고 있는 **형제**들이 그 오류로 끝난 것으로 처리된다.

5 취소는 **되감지 아니한다.** 이미 일을 낸 흐름의 효과는 되돌아가지 않으며, 취소는 앞으로 할 일을 하지 않게 할 뿐이다. 그러므로 "어느 형제가 취소되었는가"는 순서에 달려 있고, 이 문서가 그 순서를 정하지 아니한다.

6 오류 자체는 `await`(§10.3 (2b))로 드러난다 — 취소가 오류를 삼키지 아니한다.

쉬운 말로

왜 묶는가. 흐름을 만들어 놓고 잊으면 그것은 프로그램이 끝난 뒤에도 남거나, 이미 없어진 것을 만지거나, 아무도 읽지 않는 실패를 낸다. 블록이 그 수명을 쥐면 그 셋이 **문법으로** 사라진다 — 잊을 자리가 없기 때문이다.

10.5 메시지를 나중에 배달하기

1 `send`(§10.3 (3))는 액터가 메시지를 처리할 때까지 기다린다. `spawn send <액터> <메시지> [인자…]` 는 기다리지 아니하고 **우편함에 넣고 곧바로 돌아온다.**

1a `try spawn send …` 는 우편함이 가득 차 있으면 멈추지 아니하고 그것을 값으로 낸다 — 그 타입은 `result u64 mailbox_full` 이다.

1b 기다리는 `send` 를 받는 액터가 **다른 흐름이 쥔 메시지를 처리하는 중이고 그 흐름이 멈추어 있으면**(`yield` · 채널 · `await` · 기다리는 `send` 로) — 우편함에서 꺼낸 메시지이든 다른 흐름의 기다리는 `send` 이든 — 이 `send` 는 그 메시지가 **끝날 때까지 기다린** 뒤에 처리된다. 한 액터의 메시지는 한 번에 하나씩이다(4a). 그 액터를 쥔 것이 **이 `send` 를 부른 사슬 자신**이면(처리 코드가 자기에게, 또는 A → B → A 로 돌아와) 기다리지 아니하고 곧바로 처리한다 — 그 깊이는 `mailbox bounded N` 이 센다(§10.2). 흐름들이 서로를 기다려 어느 것도 끝날 수 없으면 프로그램이 멈춘다(`E-VM-DEADLOCK`).

1c `try send <액터> <메시지> [인자…]` 는 (1b) 에서 기다려야 할 자리에서 **기다리지 아니하고** 그것을 값으로 낸다 — 그 타입은 `result <처리 코드의 출력> actor_busy` 이다. 기다릴 일이 없으면 `send` 와 같고, 처리 코드의 답이 `ok` 에 든다.

쉬운 말로

전에는 (1b) 의 `send` 가 그 자리에서 처리 코드를 불렀다. 그러면 멈춘 메시지가 깨어나 자기가 읽어 둔 옛 상태로 그 일을 덮었다 — 보낸 쪽은 답을 받았는데 그 일은 사라졌다. 기다리면 그 일이 남고, 기다릴 수 없는 자리는 `try send` 로 그 사실을 값으로 받는다.

2 우편함에 든 것은 **저절로 처리되지 아니한다.** `drain <액터>` 가 그 액터의 우편함을 **넣은 차례대로** 비운다.

2a `drain` 을 부를 때 그 액터가 우편함에서 꺼낸 메시지를 **아직 처리하는 중**이면 — `schedule` 이 꺼내 놓고 끝나지 않았거나, 다른 `drain` 이 그 메시지를 돌리는 중이면 — 프로그램이 멈춘다(`E-VM-DRAIN-BUSY`). 그대로 배달하지도, 기다리지도 아니한다.

쉬운 말로

그대로 배달하면 한 액터가 두 메시지를 함께 처리하게 된다 — (4a) 가 막은 바로 그것이다. 기다리게 하면, 그 메시지의 처리 코드 안에서 부른 경우에는 기다림이 영영 끝나지 않고, `drain` 이 동료를 기다리는 일(`concurrent`)이 된다. 멈추면 그 자리가 드러나고 `drain` 의 뜻이 그대로 남는다. 비우는 자리는 그 메시지가 끝난 뒤로 옮긴다 — 예를 들어 `schedule` 을 부른 코드에서.

3 처리기가 언제 배달할지 스스로 정하지 아니하는 까닭은 **결정성** 때문이다 — 같은 프로그램이 같은 답을 내야 하고, 배달 시점이 곧 답이다.

주의 —

우편함에 넣기만 하고 비우지 아니하면 그 메시지는 **처리되지 않는다.** 이것은 잊힌 실패가 아니라 적은 대로 된 것이다 — 비우는 자리를 사람이 고른다.

4 `schedule .` 은 **모든 액터의** 대기 메시지를 더 남은 것이 없을 때까지 배달한다. `drain` 이 하나를 비운다면 이것은 전부를 비운다.

4a `schedule` 도 **한 액터 앞으로 온 메시지는 넣은 차례대로, 한 번에 하나씩** 처리한다 — `drain` 과 같다. 처리 코드가 `yield`(§10.6 (4))로 자리를 넘기거나 채널에서 멈추어도, 그 액터의 다음 메시지는 앞의 것이 **끝난 뒤에** 시작한다.

4b 서로 다른 액터의 메시지 사이, 메시지와 태스크 사이의 차례는 이 문서가 정하지 아니한다. 답이 그 차례에 기대는지는 `schedule explore_interleavings`(§6.4.13 (4))가 열린 차례를 모두 돌려 본다.

쉬운 말로

한 액터 안의 차례를 정하는 까닭은 액터가 **자기 상태를 혼자 가진다**(§10.2 (1))는 데 있다. 같은 액터의 두 메시지가 함께 진행되면 둘이 같은 상태를 번갈아 고치게 되고, 그러면 액터 안에 경합이 생긴다 — 액터를 쓰는 까닭이 사라진다. 액터 사이의 차례를 열어 두는 까닭은 그것을 정해도 얻는 것이 없고 잃는 것이 있기 때문이다: 정해 두면 그 차례에 기대는 프로그램이 시험에서 드러나지 않고, 여러 코어로 나눌 때 모든 액터가 서로를 기다려야 한다.

예제 (example) — 한 액터 안은 넣은 차례대로

소스 코드 (source)

```
module ex_schedule_order .

actor log do
  state do h u64 . end .
  proc put input x u64 . output u64 . effects state . do
    set h. expr h. * 10 + x. . .
    return h. .
  end .
  fn get output u64 . do return h. . end .
end .

export proc run output u64 . effects state .
do
  var c log spawn actor log . .
  spawn send c. put 1 . .
  spawn send c. put 2 . .
  spawn send c. put 3 . .
  spawn send c. put 4 . .
  spawn send c. put 5 . .
  schedule .
```

```
  return send c. get . .
end .
```

실행 결과 (output)

```
run() = 12345
```

10.6 흐름끼리 주고받기 — 채널

채널 (channel)

흐름 사이에 값을 넘기는 그릇. 넣은 차례대로 나오며, 담을 수 있는 수가 정해져 있다. 가득 차거나 비면 그것을 쓰는 흐름이 멈추었다가 상대가 오면 이어 간다.

1 채널 (channel) 은 흐름 사이에 값을 넣고 빼는 **차례 있는 그릇**이다. `channel [<타입>]` 으로 만들고, `chsend <채널> <값>` 으로 넣고, `chrecv <채널>` 로 뺀다.

1a `channel T` 의 타입은 `chan T` 다. `chsend` 에 넣는 값은 `T` 에 들어가야 하고(`E-TYPE-WIDTH` · `E-TYPE-SIGN`), `chsend` 는 `u64` `0` 을 낸다. `chrecv` 는 `T` 를 낸다. `chan T` 가 아닌 값에 넣거나 빼는 것은 거부된다(`E-TYPE-KIND`).

1b 채널 핸들도 `0` 이 비어 있는 핸들이다(§10.3 (2d)). `chan T` 로 선언한 이름은 리터럴 `0` 으로 시작할 수 있고, 그 밖의 수는 받지 아니한다. 핸들 `0` 에 넣거나 빼면 프로그램이 멈춘다(`E-VM-HANDLE-ZERO`).

예제 (example) — 핸들과 채널의 타입

소스 코드 (source)

```
module ex_typed_flow .

fn square input n u64 . output u64 .
  requires le n. 1000 . .
do
  return mul n. n. . .
end .

proc producer input ch chan u64 . output u64 . effects concurrent .
do
  return chsend ch. 25 . .
end .

proc consumer input ch chan u64 . output u64 . effects concurrent .
do
  return chrecv ch. . .
end .

rem 핸들은 `task u64`, 채널은 `chan u64` — 흐름이 낼 값과 채널에 담기는 값의 타입이 타입에 실린다.
export proc both output u64 . effects state concurrent .
do
  var r u64 0 .
  var ch chan u64 channel u64 . .
  task_group do
    let h task u64 spawn square 6 . .
    let g task u64 spawn consumer ch. . .
    spawn producer ch. .
    set r. add await h. . await g. . . .
  end .
```

```
  return r. .
end .
```

실행 결과 (output)

```
both() = 61
```

거부되는 예제 (rejected) — 핸들을 수로 받는다

소스 코드 (source)

```
module ex_handle_as_number .

fn square input n u64 . output u64 .
  requires le n. 1000 . .
do
  return mul n. n. . .
end .

export proc loose output u64 . effects state concurrent .
do
  var r u64 0 .
  task_group do
    let h u64 spawn square 5 . .
    set r. 1 .
  end .
  return r. .
end .
```

진단: `E-TYPE-LET`

2 채널은 **크기가 정해져 있다.** 가득 찬 채널에 넣으려 하면 넣는 흐름이 멈추고, 빈 채널에서 빼려 하면 빼는 흐름이 멈춘다. 멈춘 흐름은 상대가 오면 **다시 이어 간다.**

3 그러므로 채널 연산은 `concurrent` 효과를 낸다(§7.1) — 완결이 **다른 흐름의 진행에 달려 있기** 때문이다. 이것은 커널이 언젠가 깨워 주는 `wait` 와 다른 요구이며, 그 차이가 시그니처에 실린다.

4 `yield` 는 지금 도는 흐름이 **자리를 넘긴다.** 채널에서 멈추는 것은 이 넘김 위에 얹혀 있다.

5 모든 흐름이 채널에서 멈추어 아무도 나아가지 못하면 그것은 **정지**이며, 처리기는 그것을 말하여야 한다. 조용히 매달리지 아니한다.

쉬운 말로

왜 효과가 갈리는가. `wait` 하는 op 은 혼자 두어도 언젠가 끝나지만, `concurrent` 한 op 은 **동료가 없으면 영원히 끝나지 않는다.** 그래서 묶인 흐름이 하나뿐인데 동료를 요구하면 그 프로그램은 어떤 순서로도 끝날 수 없고, 처리기는 그것을 **번역할 때** 거절한다 — 돌려 보고 매달리는 것을 기다리지 아니한다.

10.7 어디까지 쓸 수 있는가 — `build profile`

1 프로그램은 자기가 어느 자리에서 도는지를 `build profile <이름> .` 으로 적을 수 있다. 적으면 처리기는 그 자리가 감당하지 못하는 동시성을 **번역할 때** 거절한다.

2 프로파일은 다음 넷이 전부이며, 그 밖의 이름은 거부된다(`E-PROFILE-UNKNOWN`).

표 71 — 프로파일과 그것이 여는 등급

프로파일	여는 등급	무엇을 뜻하나
`freestanding`	0	운영체제가 없다. 동시성을 쓰지 아니한다

`embedded`	1	정해진 일감을 나누어 도는 데까지 — 흐름이 **정적으로** 정해진다
`native`	2	흐름을 만들고 채널로 주고받는다
`server`	3	액터까지 — 상태를 가진 흐름이 메시지로 산다

3 프로그램이 쓰는 것이 프로파일이 여는 등급보다 높으면 거부된다(`E-PROFILE-LEVEL`). `actor` 를 선언하는 것만으로 3 등급이며, `spawn` 과 `task_group` 은 1 등급이다.

4 **프로파일을 안 적으면 게이팅하지 아니한다.** 적은 사람만 그 약속을 지게 된다 — 적지 않은 프로그램은 자기가 어디서 도는지 말하지 않은 것이고, 말하지 않은 것을 처리기가 지어내지 아니한다.

> **쉬운 말로**
>
> 왜 낱말 하나로 등급을 판정하지 아니하는가. `send` 는 액터에도 채널에도 쓰이고 `drain` 은 사용자가 지은 op 의 이름일 수도 있다. 낱말만 보고 등급을 올리면 **엉뚱한 프로그램을 고발한다.** 그래서 3 등급은 `actor` **선언**으로만 판정한다 — 한 낱말이 두 뜻이면 그 낱말로 판정하지 아니한다.

10.8 기계가 감당하는 것 — `build tier`

1 §10.7 의 프로파일이 "어느 자리에서 도는가"를 말한다면, `build tier <이름>` . 은 "이 기계가 무엇을 **감당하는가**"를 말한다. 둘은 다른 축이다 — 하나는 쓰는 짜임을 가르고, 다른 하나는 **효과**를 가른다.

2 등급은 넷이며 닫힌 집합이다. 그 밖의 이름은 거부된다(`E-BUILD-TIER`).

표 72 — 등급과 그것이 감당하는 효과

등급	어떤 기계	새로 감당하는 효과
`t0`	협력 바닥 — 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**

3 선언한 등급이 감당하지 못하는 효과를 쓰면 **번역할 때** 거부된다(`E-TIER-EFFECT`). 이것은 실행 중에 느려지는 문제가 아니라 **애초에 그 기계에 실을 수 없는 것**이다.

4 효과는 부르는 사슬을 타고 올라오므로(§7.1), 거절은 **부르는 자리**에서 난다 — 효과가 처음 생긴 자리가 아니다.

5 **안 적으면 막지 아니한다.** 적지 않은 프로그램의 기본은 `t3` 이며, 곧 아무것도 막히지 않는다. 게이트는 **고르는 것**이다.

> **쉬운 말로**
>
> 왜 두 축을 하나로 합치지 아니하는가. 프로파일은 짜임(액터·흐름·채널)을 가르고 등급은 효과(무더기·잠금·원자)를 가른다. 운영체제가 있는 작은 기계도 있고, 없는 큰 기계도 있다. 둘을 한 낱말로 묶으면 그 둘이 어긋나는 기계를 **말할 수 없게** 된다. 축이 둘이면 낱말도 둘이다.

10.9 나란히 도는 것이 어긋나는 자리

1 흐름을 만드는 것(`spawn`)은 **묶는 자리 안에서만** 할 수 있다(`E-SPAWN-SCOPE`, §10.4). 아무도 기다려 주지 않는 흐름은 만들지 아니한다.

2 묶는 자리가 흐름을 **하나만** 만들고 그 흐름이 짝을 기다리면 거부된다 (`E-CONC-ALONE`) — 짝이 없는 기다림은 영영 끝나지 아니한다.

3 묶인 흐름들이 모두 받기만 하고 **아무도 보내지 않으면** 거부된다 (`E-CONC-DEADLOCK`). 실행해 보아야 아는 것이 아니라 **적힌 것만 보아도** 아는 교착이므로, 그 자리에서 막는다.

3a 액터에게 그 액터가 **다루지 않는 말**을 보내는 것은 적합하지 아니하다 (`E-IR-UNDEF`). 다루는 말은 그 액터의 선언에서 찾으며, 이름이 같은 다른 액터의 것을 **말없이 빌려 쓰지 아니한다.**

4 끝이 없는 통로(`channel … unbounded`)는 받아들이지 아니한다 (`E-CHAN-UNBOUNDED`). 크기가 정해진 통로를 쓴다.

5 나누어 가지는 상태 타입(`lock · rwlock · shared_read`)은 아직 짓지 아니하였다 (`E-LOCK-NOTYET`).

> **쉬운 말로**
>
> (4) 는 **짓지 않기로 정한 것**이다(RFC-0150 Q6 ©, 2026-10-15). 끝이 없는 통로는 메모리를 얼마나 쓸지 위에 적을 수 없는 값이라, 비용이 글에 보인다는 원칙(§1.3)과 부딪친다. 차는 통로는 보내는 쪽을 멈추게 하고, 그 멈춤은 글에서 보인다.

> **주의 —**
>
> (3) 은 모든 교착을 잡지 못한다 — 적힌 것만 보고 아는 것만 잡는다. 잡지 못하는 교착이 있다는 사실은 이 규칙이 약하다는 뜻이 아니라, **이 규칙이 무엇을 약속하는지**를 분명히 하는 것이다(§8.6).

A 부록 A — 문법 요약 (Annex A: Grammar summary)

1 이 부록은 §6 이 정한 문법 가운데 **자주 찾는 것을 모은 참고 자료**다. 규범은 본문이며, 이 부록과 본문이 어긋나면 본문이 옳다.

1a **낱말 목록(A.1)과 전체 문법(A.10)은 망라한다** — A.1 에 없는 것은 낱말이 아니고, A.10 의 생성 규칙이 읽지 못하는 낱말의 나열은 이 언어의 소스가 아니다. A.2 – A.5 와 A.7 은 자주 쓰는 모양을 보기 좋게 모은 것이며 모양을 다 담지 아니한다.

1b A.10 이 정하는 것은 **모양**뿐이다(§1.5 (4)). 모양이 맞는 것 가운데 어떤 것이 적합한지는 본문의 조항이 정한다.

A.1 낱말

1 다음이 낱말의 전부다. 이 목록에 없는 것은 낱말이 아니다.

> **문법 틀 (grammar shape) — 낱말 목록 — 이 목록이 전부다**

```
actor     break     case      continue  contract  cycle     do
drop      else      end       enum      expect    export    expr
extern    false     fn        for       guard     if        keep
let       lit       match     module    newtype   next      none
proc      range     repeat    return    satisfies send      set
spawn     state     step      struct    test      trait     true
try       type      unsafe    use       var       while
```

2 예약된 낱말은 이름이 될 수 없다(§6.1.2). 이 목록의 낱말은 모두 이 문서가 다루며, 다루지 않는 채 예약만 해 둔 낱말은 **없다**.

> **참고 (informative)**
>
> 이 목록이 한 화면에 담기는 것은 우연이 아니라 규율이다(§6.1.2). 낱말을 더하려면 그만한 값어치를 보여야 한다.

A.2 문장의 모양

1 아래는 자주 쓰는 모양을 모은 것이다. <…> 는 채워 넣는 자리다. 블록을 가진 선언도 `end` 뒤의 점으로 닫힌다(§6.1.6 (2b)).

> **문법 틀 (grammar shape) — 선언**

```
module <이름> .

use <모듈이름> .

type <이름> <타입> .
newtype <이름> <타입> .

struct <이름> do
  <칸이름> <타입> .
end .

enum <이름> do
  <갈래이름> .
end .

export fn <이름> input <이름> <타입> . output <타입> .
  requires <조건> .
  ensures <조건> .
```

```
do
  <문장들>
end .

proc <이름> input <이름> <타입> . output <타입> . effects <효과들> .
do
  <문장들>
end .
```

A.3 문장

문법 틀 (grammar shape) — 문장

```
let <이름> <타입> <식> .
var <이름> <타입> <식> .
let <이름> <타입> <식> else <빠져나가는 문장> .
set <자리> <식> .

if <조건> do <문장들> end .
if <조건> do <문장들> end else do <문장들> end .
while <조건> do <문장들> end .
for <이름> <원천> do <문장들> end .
repeat <이름> <타입> <횟수> do <문장들> end .
range <이름> <타입> <처음> <끝> do <문장들> end .
cycle <이름> <타입> <처음> while <조건> next <다음 값> do <문장들> end .
guard <조건> else <빠져나가는 문장> .

return <식> .
break [<수>] .
continue [<수>] .
```

1 이 틀의 <식> · <조건> · <원천> · <자리> 는 **제 점까지 든 폼**이다 — 변수면 `a.`, 부름이면 `add a. 1 .`, 리터럴이면 `7` 이다(§6.1.5). 줄 끝의 점은 그 문장을 닫는다. 블록을 가진 문장도 `end` 뒤의 점으로 닫히고, `else` 로 이어진 사슬은 끝에 점 하나다. <빠져나가는 문장> 도 제 점을 갖는다(`guard gt a. 0 . else return 0 . .`). 이 문장들의 생성 규칙은 A.10.3 에 있고, 뜻은 §6.5.1 · §6.5.3.1 · §6.5.4 가 정한다.

A.4 갈래·시험·액터

문법 틀 (grammar shape) — 갈래를 가르기 · 시험 · 액터

```
match <값> do
  case <갈래> do <문장들> end .
  case <갈래> do <문장들> end .
end .

test <이름>
do
  expect <조건> .
end .

actor <이름> do
  state do
    <칸이름> <타입> .
  end .
```

```
  proc <이름> output <타입> . effects state .
  do
    <문장들>
  end .
end .
```

A.5 식

문법 틀 (grammar shape) — 식

```
rem 전위 — 이름이 열고 점이 닫는다. 우선순위가 없다
add <a> <b> .        sub <a> <b> .        mul <a> <b> .
div <a> <b> .        mod <a> <b> .
eq <a> <b> .         ne <a> <b> .         lt <a> <b> .
le <a> <b> .         gt <a> <b> .         ge <a> <b> .
and <a> <b> .        or <a> <b> .         not <a> .
len <슬라이스> .      idx <슬라이스> <번호> .
widen <타입> <값> .   narrow <타입> <값> .

rem 변수 — 이름에 점을 붙인다. 리터럴과 true · false · none 에는 점이 없다
<이름>.

rem 중위 — expr 섬 안에서만. 섬도 제 점으로 닫힌다
expr <a> + <b> * <c> .
```

A.6 리터럴의 어휘 문법

1 이 절은 §6.1.4 가 산문으로 규범한 리터럴을 **생성 규칙으로** 다시 적는다. 산문과 이 규칙이 갈리면 산문이 이긴다 — 이 절은 요약이지 새 규범이 아니다.

2 표기: `{ }` 는 0 회 이상, `[ ]` 는 선택, `¦` 는 택일, `'…'` 는 그대로의 글자다. 대문자 이름은 렉서가 내는 토큰이고, 소문자 이름은 그 토큰을 이루는 조각이다.

문법 틀 (grammar shape) — 수 리터럴

```
NUMBER  = [ sign ] , ( float ¦ hex ¦ bin ¦ dec ) ;
sign    = '+' ¦ '-' ;
sep     = '_' ;
digit   = '0'…'9' ;
hexd    = digit ¦ 'a'…'f' ¦ 'A'…'F' ;
bind    = '0' ¦ '1' ;

dec     = digit , { [ sep ] , digit } ;
hex     = '0' , ( 'x' ¦ 'X' ) , hexd , { [ sep ] , hexd } ;
bin     = '0' , ( 'b' ¦ 'B' ) , bind , { [ sep ] , bind } ;

float   = dec , '.' , dec , [ dexp ]
        ¦ dec , dexp
        ¦ hex , [ '.' , hexd , { [ sep ] , hexd } ] , hexp ;
dexp    = ( 'e' ¦ 'E' ) , [ sign ] , dec ;
hexp    = ( 'p' ¦ 'P' ) , [ sign ] , dec ;
```

3 **부호는 붙여야 부호다.** `-5` 는 리터럴 하나이고 `- 5` 는 아니다. `expr` 섬(§6.3) **밖**에서 숫자에 붙은 부호는 값의 일부이며, 섬 **안**에서 `-` 는 연산자다. 그러므로 폼 자리의 `g 10 -3` 은 인자 **둘**이지 뺄셈이 아니다. 그 갈림을 띄어쓰기가 아니라 **섬이** 정한다.

4 **구분자는 자릿수 사이에만 온다.** 위 규칙의 `[ sep ] , digit` 이 그 뜻이다 — 맨 앞 (`_1`)·맨 뒤(`1_`)·진법 표시 바로 뒤(`0x_1`)에는 올 수 없다. 구분자는 모든 진법과 부동소수점 수의 **모든 부분**(정수부·소수부·지수부)에서 쓸 수 있다.

5 **`float` 이 `dec` 보다 먼저 온다.** 점도 지수도 없으면 `dec` 로 읽히고, 점이나 지수가 있으면 `float` 이다. 점 **앞뒤 모두** 자릿수가 있어야 하므로(§6.1.4 (2)) `1.` 은 부동소수점 수가 아니라 정수 `1` 뒤에 폼을 닫는 점이다.

6 **앞의 0 은 팔진이 아니다.** `dec` 는 `0` 으로 시작할 수 있고 값은 십진이다.

> **문법 틀 (grammar shape) — 문자·문자열 리터럴**

```
CHAR    = [ prefix ] , "'" , ( charchar ¦ escape ) , "'" ;
STRING  = [ prefix ] , '"' , { strchar ¦ escape } , '"'
        ¦ 'r' , '"' , { rawchar } , '"' ;   (* 날 문자열 – 이스케이프 없음, §6.1 (13a) *)

prefix  = 'u' ¦ 'U' ;
charchar= ? "'" 도 '\' 도 아닌 글자 하나(UTF-8) ? ;
strchar = ? '"' 도 '\' 도 아닌 바이트 하나 ? ;
rawchar = ? '"' 도 줄바꿈도 아닌 바이트 하나 ? ;
escape  = ? ⟦§6.1.4⟧ 의 «이스케이프 – 닫힌 집합 열넷» 표에 있는 것 ? ;
```

> **문법 틀 (grammar shape) — 텍스트 리터럴과 여러 줄 주석**

```
TEXTLIT = 'text' , sp , { sp } , tag , [ sp , { sp } , proc ] , { sp } , [ rem ] , nl ,
          { line , nl } ,
          tag , { sp } , ( nl ¦ eof ) ;
NOTE    = 'note' , sp , { sp } , tag , { linech } , nl ,
          { line , nl } ,
          tag , { sp } , ( nl ¦ eof ) ;

tag     = ( letter ¦ '_' ) , { letter ¦ digit ¦ '_' } ;
proc    = 'u' ¦ 'U' ;
rem     = 'rem' , { linech } ;
line    = ? 닫는 줄이 아닌 줄 ? ;
linech  = ? 줄바꿈이 아닌 바이트 ? ;
sp      = ' ' ¦ 탭 ¦ CR ;
nl      = LF ;
eof     = ? 파일의 끝 ? ;
letter  = 'a'…'z' ¦ 'A'…'Z' ;
```

6a 닫는 줄은 **줄의 맨 첫 칸에서** 여는 줄과 같은 `tag` 로 시작하고, 그 바로 뒤가 `letter · digit · '_'` 가 아닌 줄이다. 그러므로 들여 쓴 `tag` 와 `tag` 보다 긴 낱말로 시작하는 줄은 `line` 이다. 위 규칙의 두 번째 `tag` 는 첫 번째와 **같은 글자열**이어야 한다 — 이 조건은 생성 규칙만으로는 적을 수 없어 여기 글로 적는다.

6b `TEXTLIT` 의 값은 `line` 들을 그 사이의 `nl` 과 함께 이은 바이트열이다. **마지막 `line` 뒤의 `nl` 은 값에 들지 아니한다.** `line` 이 하나도 없으면 값은 빈 바이트열이다. 본문은 이스케이프를 풀지 아니한다 (§6.1.4 (29)).

6c 닫는 줄에서 `tag` 와 `{ sp }` 뒤에 `nl` 도 `eof` 도 아닌 것이 오면 번역이 거부된다 (`TEXTLIT` 은 `E-TEXTLIT-TAIL`, `NOTE` 는 `E-NOTE-TAIL`). 여는 줄에서 `tag` 뒤에 `proc` 도 `rem` 도 아닌 것이 오면 거부된다(`E-TEXTLIT-OPENER`). `NOTE` 의 여는 줄은 `tag` 뒤를 줄 끝까지 읽지 아니한다. `note` 뒤에 `tag` 가 없으면 거부된다 (`E-NOTE-TERM`) — `TEXTLIT` 은 같은 자리에서 `E-TEXTLIT-TERM` 이다.

7 접두사는 **닫힌 집합 둘**이다. 그 밖의 글자를 리터럴 앞에 붙이면 번역이 거부된다 (`E-STR-PREFIX`). 접두사가 값의 원소를 무엇으로 볼지 정한다(§6.1.4 (27)).

8 문자 리터럴은 **한 칸에 들어가야 한다.** 접두사가 정한 원소 하나를 넘으면 거부되며 (E-CHAR-WIDTH), 빈 것도 거부된다(E-CHAR-EMPTY).

> **참고 (informative)**
>
> 이스케이프의 집합을 여기에 다시 적지 않은 것은 뜻이 있다. 같은 목록이 두 곳에 있으면 둘은 반드시 갈리고, 갈린 뒤에는 어느 쪽이 규범인지 아무도 모른다. 집합은 §6.1.4 의 표 한 곳에만 있다.

A.7 머리 낱말과 닫개

1 폼은 **이름** 하나로 열려 **닫개** — 점 — 하나로 닫힌다(§6.1.5 · §6.1.6). 다음 표가 머리 낱말마다 어떤 모양을 갖는지를 모은 것이다. 닫개는 어느 줄에서나 점이다 — 블록을 가진 폼은 `end .` 으로 끝난다. 그래서 닫개 칸은 따로 채우지 않고, 모양의 끝에 적힌 점이 그것이다.

2 표기: `<…>` 는 채워 넣는 자리, `*` 는 0 회 이상, `[…]` 는 선택이다.

> **문법 틀 (grammar shape) — 머리 · 모양 · 닫개**

머리	모양	닫개
module	module <이름> .	
use	use <이름> from "<경로>" [as <별칭>] .	
type	type <이름> <타입> .	
newtype	newtype <이름> <타입> .	
struct	struct <이름> do <칸>* end . (칸 = <이름> <타입> .)	
enum	enum <이름> do <갈래>* end . (갈래 = <이름> [<칸>*] .)	
trait	trait <이름> do <서명>* end . (서명 = <이름> <절>*)	
actor	actor <이름> do <state·절·op>* end .	
state	state do <칸>* end . (actor 안)	
contract	contract <이름> do <절>* end .	
fn / proc	[꾸밈]* fn <이름> <절>* do <폼>* end .	
test	test <이름> [schedule <절>] do <폼>* end .	
expect	expect <조건> . (시험 블록의 단언)	
input	input [comptime] <이름> <타입> .	
use (op 머리)	use <이름> <타입> . (op 이 깎아 쓰는 얼로케이터)	
output	output <타입> . ¦ output <이름> <타입> . (이름 붙은 결과 — 0 에서 시작하는 지역, §6.4.1 (3e))	
effects	effects <원자>* .	
access	access <이름> <모드> .	
parallel	parallel <이름> <모드> .	
requires / ensures	requires <조건-폼>* .	
errors	errors <갈래> [<조건-폼>] . (한 절에 오류 하나)	
tests	tests <이름>* .	
let / var	let <이름> [use <이름>. ¦ keep <이름>.] <타입> <폼> [else …] . (else 가 블록으로 끝나도 점)	
set	set <자리-폼> <폼> .	
into	pop <스택> into <자리> . (프렐류드 문형)	
if (문)	if <폼> do <폼>* end [else if <폼> do <폼>* end]* [else do <폼>* end] .	
guard	guard <폼> else <나가는-폼> . . (나가는 폼의 점, 그리고 guard 의 점)	
while	while <폼> do <폼>* end .	
for	for <이름> [mut] <폼> [if <폼>] do <폼>* end .	
repeat	repeat <이름> <타입> <폼> [if <폼>] do <폼>* end .	
range	range <이름> <타입> <폼> <폼> [step <폼>] [if <폼>] do <폼>* end .	
cycle	cycle <이름> <타입> <폼> while <폼> next <폼> [if <폼>] do <폼>* end .	
region	region <이름> <종류> do <폼>* end .	
borrow	borrow <이름> <폼> do <폼>* end .	
return	return [<폼>] .	
break	break [<수>] . (수 = 벗어날 되풀이의 겹 · 라벨은 없다)	
continue	continue [<수>] .	

```
lit                 lit <타입> do <칸초기>* end .  (칸초기 = <이름> <폼> .)
lit (나열)          lit array <길이> <타입> <값>… [_] .  ·  lit slice <타입> <값>… .
lit (matrix)        lit matrix <차원 수> <크기>… <타입> <값>… [_] .
lit (칸 채우기)     lit array <길이> <타입> do <번호> <값> . … [_ <값> .] end .
lit (SIMD)          lit vec <레인> <타입> <값>… [_] .
extern              extern fn/proc <이름> do <절>* end .  (몸이 씨)
match               match <폼> do <가지>* end [else do <폼>* end] .   (가지 = case <패턴> do <폼
>* end .)
```

2a fn/proc 머리의 <절>* 은 §6.4.1 (3a) 의 한 차례를 따른다: satisfies·lowdoc · vector·priority · comptime 입력 · 권한·영역 입력 · use · 데이터 입력 · output · effects · link·variadic · asm · asm·absorbs·reference·why · access·inplace·invalidates·parallel·reduce · requires · ensures · errors · tests · schedule (E-CLAUSE-ORDER).

2b **블록 선언**(struct·enum·trait·actor·state·contract)의 몸은 do 로 열고 end 로 닫는다 — fn 의 몸과 제어 블록과 같은 한 규칙이다. struct <이름> . 처럼 점으로 열거나 이름 뒤에서 줄만 바꾸는 꼴은 거부된다(E-STMT-NODO). 개행은 닫개가 아니므로(§6.1.6) 줄바꿈으로는 머리가 닫히지 않는다.

3 이 표는 **머리와 그 모양**을 규범한다. 모양 칸은 자주 쓰는 꼴을 적은 것이며, 정확한 규범은 각 조항의 본문이다 — 둘이 갈리면 본문이 이긴다.

4 이 목록에 없는 머리는 없다. 낱말 아닌 머리(region·borrow·into)는 프렐류드 연산이며, 그것도 §9 가 정한 이름이지 새로 지을 수 있는 것이 아니다.

A.8 없앤 낱말

1 다음은 한때 이 언어에 있었으나 **없앤** 낱말이다. 적으면 거부되며 (E-VOCAB-REMOVED), 처리기는 그 자리에서 무엇을 대신 쓰는지 말한다.

표 73 — 없앤 낱말 — 그리고 대신 쓰는 것

없앤 것	대신	왜
loop	while true do	똑같은 뜻의 두 철자였다
give	return	똑같은 뜻의 두 철자였다
unit	void	한 뜻에 두 철자 — 명세와 도구가 다르게 불렀다
calcop	fn	순수한 셈은 함수다. 그리고 «op» 은 이제 프렐류드 연산만 가리킨다
procop	proc	효과를 내는 것은 수학의 함수가 아니다
is	(적지 않는다)	장식이었다 — 아무도 읽지 않아 type h zzz u8 . 이 통과했다
as	(적지 않는다)	어디서도 무게가 없었다 — 블록 이름은 읽히고 **버려졌다**
local	(적지 않는다)	기본값이 이미 그것이다 — export 아닌 것은 밖에서 안 보인다
to	field a. b .	중위 접근을 없앴다 — 한 뜻에 철자가 넷이었다
in	field a. b . · idx a. i. .	to 의 거꾸로 철자 — 셋째 철자였다
when	(적지 않는다)	errors 절이 오류 하나만 받으므로 표식이 필요 없다

`on`	`proc`	액터 블록 안이면 이미 메시지 처리기다. 게다가 **순수/절차 비트를 우회했다**
`fail`	`return error <갈래>`	명세의 어휘에 아예 없었다 — 옛 해석기에만 살아 있었다
`;`	`.`	닫개의 **세 번째 철자**였다(⟦§6.1.6⟧)
`make`	`lit`	값 리터럴의 머리를 하나로 — 구조체·배열·벡터·슬라이스 값이 모두 `lit <타입> …` 로 시작한다
`index`	`idx`	자주 쓰는 낱말을 짧게 — 읽기 `idx a. 3 .` · 쓰기 `set idx a. 3 . v. .`
`be`	(적지 않는다)	타입의 끝은 타입 문법의 인자 수가 정한다 — 이름과 타입 사이에서 나르는 것이 없었다. 거부의 코드는 `E-LET-BE` 다(⟦§6.5.1⟧ (3a))
`.이름` (머리 표시)	맨 전위 · 괄호	머리를 여는 셋째 철자였다 — 띄어쓰기 한 칸이 닫개를 머리로 바꿨다
`not is_some x. . .`	`is_none x. .`	같은 뜻의 두 철자였다(`E-NOT-IS-SOME`)
`union` · `select` · `any` … (비트셋·레인)	`bitset_union` · `lane_select` · `lane_any` …	짧은 낱말을 저자에게 남긴다(⟦§6.3.3⟧ (1e))

2 없앤 까닭은 대개 **같은 뜻의 두 철자**이거나 **아무것도 사지 못하는 낱말**이었기 때문이다. 낱말은 목록에 오르는 값을 해야 하며, 하지 못하면 내려온다.

2a 타입 선언의 옛 머리 — `def struct` · `def enum` · `def type` · `def newtype` — 도 거부된다(`E-DEF-OLD`). 타입 선언은 제 낱말로 시작한다(§6.1.2 (3)). `def` 는 낱말이 아니므로 그 꼴이 아닌 자리에서는 저자의 이름이다. `type` 은 타입 매개변수의 종류(`input comptime t type .`)로도 쓰인다.

3 `as` 와 `to` 는 **자리 표식**으로만 남아 있다 — `use … as <별칭>` 과 `case <아래> to <위>` 에서다. 그 자리 밖에서는 낱말이 아니다.

> **쉬운 말로**
>
> 없앤 낱말을 이 문서가 적어 두는 까닭. 옛 코드나 옛 글을 읽던 사람이 그 낱말을 만나면 "내가 뭘 잘못 쓴 건가"를 묻게 된다. 여기에 적혀 있으면 그 물음이 한 줄로 끝난다 — **없앴고, 대신 이것을 쓴다.**

A.9 타입 낱말

1 타입을 적는 자리에 올 수 있는 낱말은 다음이 전부다. 여기 없는 이름은 저자가 지은 이름(§6.2.9)이거나 아무것도 아니다.

> **문법 틀 (grammar shape) — 쓸 수 있는 타입 낱말**

```
rem 수
u8   i8   u16   i16   u32   i32   u64   i64   usize  idiff   f32   f64
u128  i128
bool  void

rem 줄과 묶음
slice   array   matrix   segments   set   stack   range   vec   bitset   mask

rem 답을 담는 것
```

```
result   option

rem 흐름과 채널(⟦§10.3⟧ · ⟦§10.6⟧)
task   chan

rem 자리와 빌림
ref   mut_ref   mut   owned   region

rem 바이트 위의 창(⟦§6.2.13⟧)
view

rem 바깥과 이어지는 것
cap   mmio
fn    unsafe_fn

rem 그 밖
self
```

2 다음 낱말은 **이름으로 받되 뜻이 아직 없다.** 처리기는 그것을 말한다(`W-NOT-YET`, §4.7) — 조용히 받아 주지 아니한다.

> **문법 틀 (grammar shape) — 이름만 받는 타입 낱말**

```
byte   char   str   string   bytes_view   dyn   atomic
rng   clock   device   file_system   net   tty   process
```

2a 넷(`list·raw·addr·rng`)은 2026-09-17 판에서 이 갈래로 옮겼다. 위 (1) 의 목록에 실려 있었으나 **뜻을 정한 조항이 이 정본 어디에도 없었고**, 하강도 그 이름을 읽지 못한다 — 그 서명의 op 은 통째로 해석기로 내려간다. 뜻을 지어내는 것보다 **아직 없다고 말하는 것**이 옳다. 그 가운데 `list` 는 2026-11-12 에 §6.2.25 이 뜻을 정해 아래 (3) 으로 옮겼다.

2b `addr` 과 `raw` 는 §6.9.4 이 뜻을 정했으나 **아직 짓지 않은** 낱말이다(`addr <이름>` 은 자리를 얻고, 그것은 `unsafe` 효과다). 2026-11-12 부터 타입 자리의 그 둘은 경고가 아니라 거부다(`E-UNSAFE-UNBUILT`). 그리고 `cap rng` 의 `rng` 는 권능의 **종류**이므로 이 조항에 들지 아니한다 — 홀로 타입으로 선 `rng` 만이 이름뿐이다. 그래서 위 (1) 의 권능 종류 줄에서 `rng` 을 내렸다.

2c `rng · clock · device · file_system · net · tty · process` 일곱은 **권능의 종류**이며, `cap <종류>` 자리에서만 뜻이 있다(§7.2). 홀로 타입으로 적으면 이 갈래에 든다 — 일곱이 같은 처지이므로 **같이 말한다.** 하나만 말하고 여섯이 조용하면, 읽는 사람은 그 차이에 뜻이 있다고 여기게 된다.

3 다음 셋은 **공유 상태의 타입**이며 지금은 거절된다. 이름이 어휘에 있는 까닭은 "없는 타입"이라 말하는 것이 거짓이기 때문이다 — 명세에 있는 타입을 두고 **네 프로그램이 틀렸다** 고 말할 수는 없다.

> **문법 틀 (grammar shape) — 아직 거절되는 타입 낱말**

```
shared_read   lock   rwlock
raw   addr
c8   c16   c32   c64
b8   b16   b32   b64   b128
list
```

3a 뒤의 둘(`raw · addr`)은 §6.9.4 이 뜻을 정했으나 아직 짓지 않은 낱말이다(`E-UNSAFE-UNBUILT`) — 공유 상태 타입과 같은 까닭으로 어휘에 있다.

3b 그 뒤의 열(`c8 … c64 · b8 … b128 · list`)은 §6.2.25 이 뜻을 정했으나 아직 짓지 않은 낱말이다(`E-TYPE-UNBUILT`) — 같은 까닭으로 어휘에 있다. 저자가 그 이름의 타입을 선언하면 그 선언이 선다.

4 `unsafe_ptr` 은 타입이 아니라 **한정자**다. `mut`·`owned` 처럼 타입 앞에 붙으며, 홀로 오면 거절된다.

5 **주소 공간 한정자의 자리는 예약되어 있다.** 프로그램 메모리와 데이터 메모리가 나뉜 기계나 가속기처럼 공간마다 포인터의 크기와 뜻이 다른 실행 환경(§5.3 (4a))을 위해, 슬라이스와 참조 타입에는 «어느 주소 공간을 가리키는가» 를 적는 한정자 자리를 둔다. 그 철자와 뜻은 그런 실행 환경을 대상으로 들일 때 정한다. 지금은 기본 공간 하나뿐이며 그 자리에 오는 낱말은 없다.

> **쉬운 말로**
>
> 이 세 갈래를 갈라 적는 까닭. "쓸 수 있다"와 "이름은 안다"와 "없다"는 읽는 사람에게 **서로 다른 일**을 시킨다 — 쓰거나, 기다리거나, 다른 길을 찾거나. 셋을 한 목록에 뭉뚱그리면 그 판단을 사람이 매번 도구를 돌려 보고 해야 한다.

A.10 전체 문법

1 이 절은 소스 파일 하나의 문법을 **처음부터 끝까지** 생성 규칙으로 적는다. 표기는 §1.5 의 것이다. 본문의 조항에 같은 이름의 규칙이 있으면 이 절의 것은 그것과 **글자까지 같다.**

2 문법은 두 층이다(§1.5 (3a)). A.10.1 은 글자를 토큰으로 가르고, A.10.2 부터는 토큰의 나열을 읽는다. 공백과 주석은 A.10.1 에서 버려지고 그 뒤에는 나타나지 아니한다.

3 소스의 모양은 다음 **네 규칙**으로 선다. 이 절의 생성 규칙은 그 넷을 풀어 적은 것이다.

표 74 — 모양을 세우는 네 규칙

규칙	내용
머리	모든 문장과 선언은 머리 낱말 하나로 시작한다. 머리가 그 뒤의 모양을 정한다
블록	`do` 는 `end` 와 짝이다. `end` 는 제 `do` 만 닫는다
이름과 점	쓰는 자리의 이름은 폼을 열고, 점 하나가 가장 안쪽에 열린 폼 하나를 닫는다. 변수는 곧바로 닫힌 폼이다(a.). 문장 · 선언 · 절 · 칸도 제 머리가 열고 점이 닫는다
낱말의 자리	이름을 만드는 자리 · 타입의 자리 · 문법과 부록 D 의 표가 정한 낱말의 자리에 선 이름은 폼을 열지 아니한다(점이 없다)

4 네 규칙은 모두 **글자만 보고** 판정한다 — 선언을 읽지 않아도 나무가 선다. `op` 이 인자를 몇 개 받는지는 모양을 세우는 재료가 아니고, 나무가 선 뒤의 검사다(§6.1.5 (3)). «점» 규칙에 예외는 없다.

A.10.1 어휘 문법

```
source    ::= { separator ¦ token }
separator ::= whitespace ¦ line-comment ¦ block-comment
token     ::= keyword ¦ name ¦ path ¦ literal ¦ punctuation

punctuation ::= "." ¦ "(" ¦ ")" ¦ "," ¦ operator
operator    ::= "+" ¦ "-" ¦ "*" ¦ "/"

whitespace      ::= " " ¦ tab ¦ carriage-return ¦ newline
blank           ::= " " ¦ tab ¦ carriage-return
tab             ::= ? U+0009 ?
carriage-return ::= ? U+000D ?
newline         ::= ? U+000A ?
line-comment    ::= "rem" { line-char }
block-comment   ::= "note" blank { blank } tag { line-char } newline
                    { comment-line newline }
                    tag { blank } ( newline ¦ end-of-source )
tag             ::= name
line-char       ::= ? newline 이 아닌 글자 ?
comment-line    ::= ? 닫는 줄이 아닌 줄 ?
end-of-source   ::= ? 소스의 끝 ?
```

```
keyword ::= "actor" ¦ "break" ¦ "case" ¦ "continue" ¦ "contract" ¦ "cycle" ¦ "do"
          ¦ "drop" ¦ "else" ¦ "end" ¦ "enum" ¦ "expect" ¦ "export" ¦ "expr"
          ¦ "extern" ¦ "false" ¦ "fn" ¦ "for" ¦ "guard" ¦ "if" ¦ "keep"
          ¦ "let" ¦ "lit" ¦ "match" ¦ "module" ¦ "newtype" ¦ "next" ¦ "none"
          ¦ "proc" ¦ "range" ¦ "repeat" ¦ "return" ¦ "satisfies" ¦ "send" ¦ "set"
          ¦ "spawn" ¦ "state" ¦ "step" ¦ "struct" ¦ "test" ¦ "trait" ¦ "true"
          ¦ "try" ¦ "type" ¦ "unsafe" ¦ "use" ¦ "var" ¦ "while"

name       ::= name-start { name-char }
name-start ::= "a" … "z" ¦ "A" … "Z" ¦ "_"
name-char  ::= name-start ¦ "0" … "9"
path       ::= name "." path-part { "." path-part }
path-part  ::= name-char { name-char }

literal        ::= number ¦ char-literal ¦ string-literal ¦ text-literal

number         ::= [ sign ] ( float ¦ hex ¦ bin ¦ dec )
sign           ::= "+" ¦ "-"
digit          ::= "0" … "9"
hex-digit      ::= digit ¦ "a" … "f" ¦ "A" … "F"
bin-digit      ::= "0" ¦ "1"
dec            ::= digit { [ "_" ] digit }
hex            ::= "0" ( "x" ¦ "X" ) hex-digit { [ "_" ] hex-digit }
bin            ::= "0" ( "b" ¦ "B" ) bin-digit { [ "_" ] bin-digit }
float          ::= dec "." dec [ dec-exponent ]
                 ¦ dec dec-exponent
                 ¦ hex [ "." hex-digit { [ "_" ] hex-digit } ] hex-exponent
dec-exponent   ::= ( "e" ¦ "E" ) [ sign ] dec
hex-exponent   ::= ( "p" ¦ "P" ) [ sign ] dec

char-literal   ::= [ prefix ] single-quote ( char-char ¦ escape ) single-quote
string-literal ::= [ prefix ] double-quote { string-char ¦ escape } double-quote
                 ¦ raw-string
raw-string     ::= "r" double-quote { raw-char } double-quote
raw-char       ::= ? 큰따옴표도 줄바꿈도 아닌 바이트 하나 ?
prefix         ::= "u" ¦ "U"
escape         ::= backslash ( simple-escape
                             ¦ "x" hex-digit hex-digit
                             ¦ "u" hex-digit hex-digit hex-digit hex-digit
                             ¦ "U" hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-
digit )
simple-escape  ::= backslash ¦ double-quote ¦ single-quote
                 ¦ "a" ¦ "b" ¦ "f" ¦ "n" ¦ "r" ¦ "t" ¦ "v" ¦ "0"
single-quote   ::= ? U+0027 ?
double-quote   ::= ? U+0022 ?
backslash      ::= ? U+005C ?
char-char      ::= ? 작은따옴표도 역슬래시도 줄바꿈도 아닌 글자 하나 ?
string-char    ::= ? 큰따옴표도 역슬래시도 줄바꿈도 아닌 바이트 하나 ?

text-literal   ::= "text" blank { blank } tag [ blank { blank } processor ] { blank } [ line-comment ] newline
                   { text-line newline }
                   tag { blank } ( newline ¦ end-of-source )
processor      ::= "u" ¦ "U"
text-line      ::= ? 닫는 줄이 아닌 줄 ?
```

1 `keyword` 가운데 `rem` · `note` · `text` 로 시작하는 토큰은 없다 — 그 셋은 낱말이 아니라 주석과 텍스트 리터럴을 여는 글자열이다(§6.1.1 · §6.1.4).

2 아래 구문 문법에 `"from"` · `"input"` · `"slice"` 처럼 `keyword` 에 없는 글자열이 끝 기호로 나오면, 그것은 **그 자리에서만** 뜻을 갖는 `name` 토큰이다(문맥 낱말). 그 자리 밖에서는 보통 이름이다.

A.10.2 번역 단위와 선언

```
unit           ::= { top-form }
top-form       ::= module-decl ¦ use-decl ¦ package-decl ¦ build-decl ¦ declaration ¦ binding
module-decl    ::= "module" name "."
use-decl       ::= "use" name [ "from" string-literal ] [ "as" name ] "."
package-decl   ::= "package" name word "."
build-decl     ::= "build" name { word } "."
word           ::= name ¦ path ¦ literal
declaration    ::= { modifier } ( type-decl ¦ op-decl ¦ extern-decl ¦ actor-decl ¦ trait-decl )
                 ¦ contract-decl
                 ¦ test-decl
modifier       ::= "export" ¦ "extern" ¦ "unsafe" [ "target" name ]
type-decl      ::= ( "type" ¦ "newtype" ) name type "."
                 ¦ "struct" name "do" { field ¦ struct-attr } "end" "."
                 ¦ "enum" name "do" { variant } "end" "."
field          ::= name type "."
struct-attr    ::= ( "layout" name ¦ "align" word ¦ "mmio" number ) "."
variant        ::= name { name type } "."
op-decl        ::= ( "fn" ¦ "proc" ) op-name { clause } block "."
extern-decl    ::= ( "fn" ¦ "proc" ) op-name "do" { clause } "end" "."
op-name        ::= name ¦ path
clause         ::= "input" [ "comptime" ] name type "."
                 ¦ "use" name type "."
                 ¦ "output" [ name ] type "."
                 ¦ "effects" { name } "."
                 ¦ ( "requires" ¦ "ensures" ) [ grade ] term "."
                 ¦ "errors" name [ term ] "."
                 ¦ "tests" { name } "."
                 ¦ "satisfies" name "."
                 ¦ ( "access" ¦ "parallel" ) name name "."
                 ¦ "asm" name "." { asm-item "." }
                 ¦ other-clause-word { word } "."
grade          ::= "assume" ¦ "static" ¦ "debug"
other-clause-word ::= "reduce" ¦ "lowdoc" ¦ "vector" ¦ "priority" ¦ "inplace" ¦ "invalidates"
                 ¦ "absorbs" ¦ "reference" ¦ "why" ¦ "strlen" ¦ "link" ¦ "variadic" ¦ "schedule"
asm-item       ::= ? `asm` 절의 항목 하나 — 낱말과 리터럴의 나열(⟦§6.9⟧) ?
actor-decl     ::= "actor" name "do" { state-decl ¦ clause ¦ op-decl } "end" "."
state-decl     ::= "state" "do" { field } "end" "."
trait-decl     ::= "trait" name "do" { signature } "end" "."
signature      ::= op-name { clause }
contract-decl  ::= "contract" name "do" { clause } "end" "."
test-decl      ::= "test" name [ "schedule" { word } "." ] block "."
```

1 `extern-decl` 은 `modifier` 에 `"extern"` 이 있고 `"export"` 가 없는 선언의 꼴이다 — 몸이 씨(C)에 있으므로 블록에는 절만 든다(§6.9). 그 밖의 `fn` · `proc` 은 `op-decl` 이다.

2 `clause` 의 차례는 생성 규칙이 정하지 아니한다. 차례는 §6.4.1 (3a) 가 정한다(E-CLAUSE-ORDER).

2a `package-decl` 과 `build-decl` 의 열쇠말과 값은 §5.7 · §5.8 · §6.4.7 · §10.7 이 정한다.

3 `struct-attr` · `variant` · `signature` 의 낱낱의 제약은 §6.2.7 · §6.2.15 · §6.2.19 · §6.11 이 정한다.

4 `signature` 는 op 의 이름 하나와 그 뒤의 절들이다. 절은 절 낱말로 시작하고 제 점으로 닫히므로, 절의 점 다음에 오는 «절 낱말이 아닌 이름» 이 다음 서명의 이름이다.

5 `requires` · `ensures` 의 `term` 이 계약 · 트레이트의 이름으로 시작하면 그것은 타입에 대한 술어다 (`requires ordered t. . .` — §6.11). 꼴은 여느 폼과 같다.

A.10.3 문장

```
block          ::= "do" { statement } "end"
statement      ::= binding ¦ guard-stmt ¦ set-stmt ¦ return-stmt ¦ jump-stmt ¦ expect-stmt
                 ¦ if-statement ¦ while-stmt ¦ for-stmt ¦ repeat-stmt ¦ range-stmt ¦ cycle-stmt
                 ¦ match-stmt ¦ region-stmt ¦ borrow-stmt ¦ task-group
                 ¦ drop-stmt ¦ op-decl ¦ call-stmt
binding      ::= ( "let" ¦ "var" ) name [ alloc-clause ] [ type ] term [ fail-clause ] "."
```

```
alloc-clause ::= ( "use" ¦ "keep" ) name "."
fail-clause  ::= "else" leave "."
               ¦ "else" block
               ¦ "else" "error" name block
leave        ::= "return" [ term ] ¦ "break" [ number ] ¦ "continue" [ number ] ¦ "panic" term
guard-stmt     ::= "guard" term "else" ( leave "." ¦ block ) "."
set-stmt       ::= "set" term term "."
return-stmt    ::= "return" [ term ] "."
jump-stmt      ::= ( "break" ¦ "continue" ) [ number ] "."
expect-stmt    ::= "expect" term "."
drop-stmt      ::= "drop" name "." [ "use" name "." ] "."
call-stmt      ::= term
if-statement ::= "if" term block { "else" "if" term block } [ "else" block ] "."
while-stmt     ::= "while" term block "."
for-stmt    ::= "for"    name [ "mut" ] term [ filter ] block "."
repeat-stmt ::= "repeat" name type term [ filter ] block "."
range-stmt  ::= "range"  name type term term [ "step" term ] [ filter ] block "."
cycle-stmt  ::= "cycle"  name type term "while" term "next" term [ filter ] block "."
filter      ::= "if" term
match-stmt     ::= "match" [ "comptime" ] term "do" { case-arm ¦ "else" block } "end" [ "else" block ] "."
case-arm       ::= "case" pattern block "."
pattern        ::= word { word } ¦ word "to" word
region-stmt    ::= "region" name name block "."
borrow-stmt    ::= "borrow" name term block "."
task-group     ::= "task_group" [ name ] block "."
```

문장은 제 머리가 열고 **점 하나가 닫는다** — 블록으로 끝나는 문장도 그렇다(§6.1.6 (2b)). 점이 빠지면 번역이 거부된다(`E-DOT-MISSING`). 닫힌 것 뒤에 점을 더 적어도 거부된다(`E-CLOSER-EXTRA`).

`call-stmt` 는 `term` 그대로다 — 그 호출을 닫는 점이 곧 문장의 점이다(`print x. .` 에서 안쪽 점이 `x` 를, 바깥 점이 `print` 를 닫는다).

`set-stmt` 의 첫 `term` 은 값을 넣을 자리다 — 이름(`i.`)이거나 자리를 내는 폼(`idx a. i. .` · `field p. x .`)이다.

`statement` 를 여는 낱말(`let` · `var` · `return` · `guard` …)은 식 안에 올 수 없다. `if` 는 문장이고 값이 아니다(`E-IF-VALUE`).

머리 없이 홀로 선 `block` 은 문장이 아니다(`E-BLOCK-NOHEAD`). `pattern` 안의 이름은 폼을 열지 아니한다.

`op-decl` 이 블록 안에 서면 그것은 그 op 안에서만 보이는 op 이다(§6.10).

A.10.4 식

```
term           ::= literal ¦ "true" ¦ "false" ¦ "none"
                 ¦ call ¦ paren-term
                 ¦ struct-literal ¦ fill-literal ¦ list-literal
                 ¦ island ¦ pipe-form ¦ word-form
call           ::= ( name ¦ path ) { term } "."
paren-term     ::= "(" term ")"
struct-literal ::= "lit" type "do" { field-init } "end" "."
field-init     ::= name term "."
fill-literal   ::= "lit" "array" word type "do" { fill-item } "end" "."
fill-item      ::= ( word ¦ "_" ) term "."
list-literal   ::= "lit" list-kind { term ¦ "_" } "."
list-kind      ::= "array" word type ¦ "slice" type ¦ "vec" word type ¦ "matrix" number number { number } type
island         ::= "expr" island-item { operator island-item } "."
island-item    ::= term ¦ "(" island-item { operator island-item } ")"
pipe-form      ::= "pipe" term "do" { stage } "end" "."
stage          ::= stage-word { word ¦ term } [ "with" term ] "."
stage-word     ::= ? 파이프 단계의 낱말 — 닫힌 어휘다(⟦§6.12⟧) ?
word-form      ::= "field" term segment { segment } "."
                 ¦ "method" term { segment } name "." { term } "."
                 ¦ "payload" term name name "."
                 ¦ "send" term name { term } "."
```

```
                ¦ "spawn" "actor" path "."
                ¦ "spawn" "send" term name { term } "." "."
                ¦ "spawn" name { term } "."
                ¦ "try" term { name } "."
                ¦ "call_builtin" name { term } "."
                ¦ shaped-form
segment        ::= name ¦ number
shaped-form    ::= ? 부록 D 의 표가 낱말의 자리를 정한 기본 연산 — `cast <타입> <항> .` · `isa <항> <갈래> .`
· `size_of <타입> .` 따위 ?
```

식은 폼 하나다. `call` 은 이름이 열고 점이 닫는다 — 인자가 없으면 그 이름의 값이다(`a.`). 폼 안의 폼은 먼저 제 점으로 닫히므로 이 문법은 **선언을 읽지 않고** 식의 나무를 세운다. 인자의 수가 선언과 맞는지는 그 뒤에 본다(`E-IR-ARITY`).

`paren-term` 은 폼 하나를 감쌀 뿐 뜻을 바꾸지 아니한다(§6.1.5 (3a)).

낱말의 자리를 가진 폼은 **닫힌 어휘**다: 문장의 머리, `word-form` 에 적은 낱말, 그리고 부록 D 의 표가 모양을 적은 기본 연산(`shaped-form`). 그 밖의 `call` — 사용자가 선언한 op — 의 인자는 모두 `term` 이다. 타입을 넘길 때도 그렇다 (`max_of u64. a. b. .`).

`method` 의 op 이름은 **점으로 닫은 첫 이름**이다. 그 앞의 이름과 수는 마디이고 그 뒤는 인자다 (`method o. inner area. w. 2 .`). `payload` 의 두 낱말은 갈래와 칸의 이름이다(§6.2.19).

`island` 안의 `operator` 는 중위 연산자다(§6.3.2). 괄호는 섬의 한 토막을 묶는다.

`stage` 의 낱말과 그 인자의 꼴은 §6.12 가 정한다.

A.10.5 타입

```
type           ::= { type-qualifier } type-core
type-qualifier ::= "mut" ¦ "owned" ¦ "ref" ¦ "mut_ref" ¦ "unsafe_ptr"
type-core      ::= name ¦ path
                 ¦ "(" applied-type ")"
                 ¦ "slice" type
                 ¦ "array" word type
                 ¦ "matrix" number matrix-size { matrix-size } type
                 ¦ "option" type
                 ¦ "result" type type
                 ¦ "vec" word type
                 ¦ "segments" type
                 ¦ "task" type
                 ¦ "chan" type
                 ¦ "bitset" [ number ]
                 ¦ "bits" number
                 ¦ "cap" name
                 ¦ "region" name
                 ¦ "range" [ type ] word word
                 ¦ other-type
applied-type   ::= type ¦ ( name ¦ path ) type { type }
other-type     ::= ? ⟦§6.2⟧ 가 정하는 그 밖의 타입 꼴(`mask` · `set` · `stack` · `fn` · `unsafe_fn` ·
`mmio`) ?
matrix-size    ::= number ¦ name
```

> **참고 (informative)**
>
> `matrix-size` 의 `name` 은 op 의 `input` 에서만 쓴다 — 그 op 의 번역 시점 값 입력의 이름이다(§6.2.6.1 (3a)).

`u8` · `bool` · `void` 같은 타입 낱말(A.9)과 저자가 지은 타입의 이름은 `name` 이다.

타입은 **글자만으로 끝난다** — 닫는 표시가 없다. 생성자는 닫힌 어휘이고(`slice u8` 은 낱말 둘, `result u64 perr` 는 셋), 그 밖의 타입은 이름 하나다. 인자를 받는 사용자 타입과 인자 수가 정해지지 않은 꼴은 괄호로 싼다 (`(box u64)` — §6.5.1 (3b)).

타입 안의 이름과 수는 폼을 열지 아니한다 — 점이 없다(`array n u8`).

4 개수를 받는 생성자(`array` · `vec` · `matrix`)는 개수를 먼저, 원소 타입을 맨 뒤에 적는다. 그래서 원소가 다시 생성자여도 괄호가 들지 않는다 — `array 3 array 4 u8`(§6.2.6 (1a) · (1d)).

B 부록 B — 진단 목록 (Annex B: Diagnostics)

1 이 부록은 처리기가 낼 수 있는 진단 식별자를 모은다. 식별자는 안정하며, 한 번 정해진 뜻은 바뀌지 아니한다(§5.4).

> **참고 (informative)**
>
> 이 목록은 **생성된 것**이다. 사람이 손으로 고치지 아니한다 — 손으로 적은 목록은 언어가 자라면 곧 낡기 때문이다. 목록이 실제 처리기와 어긋나면 검사기가 실패한다.

B.1 실행 중 트랩 (E-VM-…)

1 프로그램이 도는 중에 조건이 깨져 즉시 멈춘 것. 모두 56 가지다.

> **문법 틀 (grammar shape) — 실행 중 트랩 식별자**

```
E-VM-ALIGN  E-VM-ANALYSIS  E-VM-ARG
E-VM-ARITY  E-VM-ASM  E-VM-AWAIT
E-VM-BOUNDS  E-VM-BOXPOOL  E-VM-BSETPOOL
E-VM-BUDGET  E-VM-CAST  E-VM-CHAIN-LIMIT
E-VM-CHAN  E-VM-CHAN-EMPTY  E-VM-CHAN-FULL
E-VM-CONTRACT  E-VM-COPY  E-VM-CSTR
E-VM-DANGLING  E-VM-DEADLOCK  E-VM-DEPTH
E-VM-DIV0  E-VM-DRAIN-BUSY  E-VM-ERR
E-VM-EXCL  E-VM-EXTERN  E-VM-FD
E-VM-FHANDLE  E-VM-FIELD  E-VM-FILEPOOL
E-VM-FMODE  E-VM-FWHENCE  E-VM-HANDLE-ZERO
E-VM-MAILBOX-FULL  E-VM-MBOX  E-VM-MMIO
E-VM-NODE-KIDS  E-VM-NONE  E-VM-OOM
E-VM-OVERFLOW  E-VM-PANIC  E-VM-REACTOR
E-VM-READONLY  E-VM-RECPOOL  E-VM-RESERVE
E-VM-SCHED-LOOP  E-VM-SHANDLE  E-VM-SHIFT
E-VM-SOCKPOOL  E-VM-STACK  E-VM-STACK-BUDGET
E-VM-STKPOOL  E-VM-TYPE  E-VM-UNDEF
E-VM-UNSUP  E-VM-VIEW
```

B.2 경고 (W-…)

1 번역은 되지만 저자가 봐야 하는 것. 모두 20 가지다.

> **문법 틀 (grammar shape) — 경고 식별자**

```
W-BUILTIN-APP  W-CBE-SLOW  W-COL0
W-CONFIG-DEPENDS  W-CONTRACT-IGNORED  W-EFFECT-OVER
W-ERRORS-UNRAISED  W-EXPORT-HIDDEN  W-EXPORT-NOSYM
W-FLOAT-UNTESTED  W-NOT-YET  W-PAR-OK
W-POOL-LOOP  W-PROC-PURE  W-RESULT-DISCARD
W-RFC-PENDING  W-TEST-NOT-RUN  W-UNBOUND
W-USE-EXTERNAL  W-VAR-NEVER-SET
```

B.3 번역 잘못 (E-…)

1 적합하지 않은 프로그램이라 번역이 성공하지 않는 것. 모두 369 가지다.

> **문법 틀 (grammar shape) — 번역 잘못 식별자**

```
E-ABI-TARGET-ALIGN  E-ABSORB-IMPURE  E-ABSORB-NOCONTRACT
E-ABSORB-NOREF  E-ABSORB-NOWHY  E-ABSORB-PLACE
```

```
E-ABSORB-SCOPE  E-ACCESS-MODE  E-ACTOR-FAILURE
E-ACTOR-FIELD  E-ACTOR-MAILBOX  E-ACTOR-STATE-REF
E-ACTOR-UNINIT  E-ALLOC-AMBIGUOUS  E-ALLOC-CAPACITY-MARK
E-ALLOC-NESTED  E-ALLOC-NOCAP  E-ALLOC-NOROOT
E-ALLOC-NOSOURCE  E-ALLOC-OUTLIVES  E-ALLOC-SHARED
E-ALLOC-TASK  E-ALLOC-USING-UNUSED  E-ARRAY-SET
E-ASM-BIND  E-ASM-BODY  E-ASM-NOCAP
E-ASM-NOEFFECT  E-ASM-NOUNSAFE  E-ASM-OPERAND
E-ASM-OPTLIE  E-ASM-TARGET  E-ASM-TARGET-UNKNOWN
E-ASM-UNBOUND  E-ASM-UNUSED  E-ATOMIC-NOCAP
E-ATOMIC-ORDER  E-ATOMIC-TARGET  E-BIND-ELSE
E-BITOP-TYPE  E-BITSET-ADD  E-BLOCK-NOHEAD
E-BLOCK-UNCLOSED  E-BORROW-ESCAPE  E-BORROW-EXCL
E-BORROW-FIELD  E-BOUND-UNSAT  E-BRAND-REUSED
E-BUILD-MODE  E-BUILD-TIER  E-BUILTIN-BARE
E-BUILTIN-NAME  E-CAP-FORGE  E-CAP-KIND
E-CAP-LOCAL  E-CAP-MISSING  E-CAP-NOHOST
E-CAP-OUTPUT  E-CFG-UNKNOWN-PROP  E-CHAN-UNBOUNDED
E-CHAR  E-CHAR-EMPTY  E-CHAR-NEWLINE
E-CHAR-UNTERM  E-CHAR-WIDTH  E-CLAUSE-ORDER
E-CLOSER-EXTRA  E-COLLECT-FULL  E-COMPTIME-ARG
E-COMPTIME-NONCONST  E-CONC-ALONE  E-CONC-DEADLOCK
E-CONFIG-DEPENDS  E-CONFIG-TYPE  E-CONFIG-UNDEF
E-CONST-NOTCOMPTIME  E-CONTRACT-DEAD  E-CONTRACT-IMPOSSIBLE
E-CONTRACT-MODE  E-CONTRACT-UNDEF  E-CONTRACT-UNSAT
E-COPY-LEN  E-CTRL-NODO  E-DEF-OLD
E-DOT-DOUBLE  E-DOT-MISSING  E-DROP-USE
E-EFFECT  E-EFFECT-CALC  E-EFFECT-DUP
E-EFFECT-MISSING  E-EFFECT-NO-CAP  E-EFFECT-NONE-MIX
E-EFFECT-PURITY  E-EFFECT-REDUNDANT  E-EFFECT-UNDEF
E-ENS-UNDEF  E-ENS-UNSUP  E-ENTRY-CAP
E-ENTRY-OUTPUT  E-ENTRY-PARAMS  E-ENUM-ARITY
E-ENUM-DOT  E-ENUM-FIELD  E-ENUM-INFINITE
E-ENUM-LAYOUT  E-ENUM-NOFIELD  E-ENUM-NOVARIANT
E-ENUM-NUMBER  E-ENUM-PAYLOAD  E-ENUM-UNCHECKED
E-ENUM-VARIANT  E-ERR-UNDECLARED  E-ERR-UNDEF
E-ERRORS-STATE  E-ESCAPE  E-EXCL
E-EXCL-INPLACE  E-EXCL-MOVED  E-EXPECT-PLACE
E-EXPR-CHAIN  E-EXPR-UNARY  E-FFI-BODY
E-FFI-CAPKIND  E-FFI-LINK  E-FFI-NOCAP
E-FFI-NOEFFECT  E-FFI-NOUNSAFE  E-FFI-TYPE
E-FIELD-FORM  E-FIELD-GLUED  E-FIELD-MARK
E-FLOAT-NOFLOAT  E-FLOAT-NOTYET  E-FN-CAP
E-FN-NOTEXPORT  E-FN-UNDEF  E-FN-VALUE
E-FOLD-OP  E-FOLD-ORDER  E-FOR-HEAD
E-FOR-OLD  E-FOR-STEP  E-FORM-UNEXPECTED
E-FRAME-RECURSIVE  E-FRAME-SIZE  E-GROUP-UNCLOSED
E-GUARD-FALLTHROUGH  E-HEAD-NOT-AN-OP  E-HEAP-NOCAP
E-HEAP-NOHOST  E-IF-VALUE  E-IMMUTABLE
E-INCLUDE-PLACE  E-INPLACE-FORM  E-INPLACE-UNPROVEN
E-INT128-PLACE  E-INT128-TARGET  E-INVALIDATES-FORM
E-IR-ARITY  E-IR-EXTRA  E-IR-LIMIT
E-IR-LOCALS  E-IR-SURFACE  E-IR-TYPE-SPLIT
E-IR-UNDEF  E-IR-UNSUP  E-ISR-CALLED
E-ISR-EFFECT  E-ISR-OUTPUT  E-ISR-PARAMS
E-LET-BE  E-LET-NOTYPE  E-LET-NOVALUE
E-LEX-UTF8  E-LINK-NAME  E-LIT-COUNT
```

```
E-LIT-ESCAPE  E-LIT-IDENTITY  E-LIT-INDEX
E-LIT-RANGE  E-LIT-UNBUILT  E-LIT-USING
E-LOCAL-CAPTURE  E-LOCAL-EXPORT  E-LOCAL-PLACE
E-LOCK-NOTYET  E-LOOP-COUNT  E-LOOP-OUTSIDE
E-MATCH-ARITY  E-MATCH-INEXHAUSTIVE  E-MATCH-ORBIND
E-MATCH-REDUNDANT  E-MATCH-UNDEF  E-MATRIX-INDEX
E-MATRIX-SHAPE  E-MATRIX-TYPE  E-MATRIX-USE
E-METHOD-RECV  E-METHOD-UNDEF  E-MMIO-BASE
E-MMIO-BYVALUE  E-MMIO-FIELD  E-MMIO-NOBASE
E-MMIO-NOCAP  E-MMIO-NOEFFECT  E-MMIO-NOHOST
E-MMIO-PERM  E-MMIO-PLAIN  E-MONO-FIXPOINT
E-MONO-NOTYPE  E-MREF-SLICE  E-NAME-ASCII
E-NAME-BUILTIN  E-NAME-CLAUSE  E-NAME-COLLISION
E-NAME-DOTTED  E-NAME-DUP  E-NAME-KEYWORD
E-NAME-QUALIFIER  E-NAME-SCOPE  E-NAME-SHADOW
E-NAME-VENDOR-RESERVED  E-NAME-WILDCARD  E-NEST-DEPTH
E-NOT-IS-SOME  E-NOTE-TAIL  E-NOTE-TERM
E-NOTE-UNTERM  E-NUM-EMPTY  E-NUM-SEP
E-NUM-SUFFIX  E-OPT-DEPENDS  E-OPT-TYPE
E-OPT-UNUSED  E-OWN-BARE  E-OWN-INCOMPLETE
E-OWN-JOIN  E-OWN-MOVED  E-OWN-NODE-KIDS
E-OWN-NODE-REF  E-OWN-PARTIAL  E-PAR-ASSOC
E-PAR-CARRY  E-PAR-FLOAT  E-PAR-IDENTITY
E-PAR-NOLOOP  E-PAR-READ  E-PAR-WRITE
E-PAREN-ESCAPE  E-PAREN-STRAY  E-PAREN-UNCLOSED
E-PARSE-LIMIT  E-PIPE-CONTEXT-ARG  E-PIPE-CONTEXT-TYPE
E-PIPE-NO-TERMINAL  E-PIPE-PRED  E-PIPE-STAGE
E-PIPE-WITH  E-PKG-DUP  E-PKG-FORM
E-PKG-KEY  E-PKG-NONAME  E-PKG-NOVERSION
E-PKG-VERSION  E-POOL-BOUND  E-PROFILE-LEVEL
E-PROFILE-UNKNOWN  E-REGION-ESCAPE  E-REGION-KIND
E-REGION-UNDEF  E-REQ-UNDEF  E-REQ-UNSUP
E-RESERVE-NOCAP  E-RESERVE-NOEFFECT  E-RESERVE-NOHOST
E-RESULT-NAMED  E-RESULT-NOZERO  E-RESULT-PLACE
E-RETURN-PARTIAL  E-SHIFT-RANGE  E-SHIFT-SIGNED
E-SPAWN-SCOPE  E-STACK-BUDGET  E-STMT-ELSE
E-STMT-NODO  E-STR-ESCAPE  E-STR-NEWLINE
E-STR-PREFIX  E-STR-UNTERM  E-STRUCT-CYCLE
E-STRUCT-REF-UNMANAGED  E-TARGET-INTRIN  E-TARGET-ISET
E-TARGET-LEAF  E-TEST-FAIL  E-TEXTLIT-OPENER
E-TEXTLIT-TAIL  E-TEXTLIT-TERM  E-TEXTLIT-UNTERM
E-TIER-EFFECT  E-TOPLEVEL  E-TRAIT-EFFECT
E-TRAIT-MISSING  E-TRAIT-RECV  E-TRAIT-SIG
E-TRAIT-UNDEF  E-TRY-NORESULT  E-TYPE-ALIGN
E-TYPE-ARG  E-TYPE-ARGMUT  E-TYPE-ARRAY
E-TYPE-BITCAST  E-TYPE-COLLECT  E-TYPE-COND
E-TYPE-CYCLE  E-TYPE-CYCLE-LIMIT  E-TYPE-DECL
E-TYPE-FIELD  E-TYPE-INSTANCE  E-TYPE-ITER
E-TYPE-KIND  E-TYPE-LANES  E-TYPE-LAYOUT
E-TYPE-LET  E-TYPE-LIMIT  E-TYPE-LOGICAL
E-TYPE-MASK  E-TYPE-MIX  E-TYPE-MUT
E-TYPE-NOMINAL  E-TYPE-NONZERO  E-TYPE-ORDER
E-TYPE-RANGE  E-TYPE-REF  E-TYPE-REFVAL
E-TYPE-RETMUT  E-TYPE-RETURN  E-TYPE-ROWNAME
E-TYPE-SET  E-TYPE-SIGN  E-TYPE-STRUCT
E-TYPE-TRY  E-TYPE-UNBUILT  E-TYPE-UNDEF
E-TYPE-UNKNOWN  E-TYPE-VAR  E-TYPE-WIDTH
```

```
E-TYPE-WRAP  E-UNSAFE-UNBUILT  E-UNSAFE-UNDECLARED
E-UNSAFE-UNUSED  E-UPTR-TYPE  E-USE-ALIASED
E-USING-DUP  E-USING-FORM  E-USING-OLD
E-USING-UNRESOLVED  E-VALUE-DISCARDED  E-VEC-QUAL
E-VEC-SPLAT  E-VIEW-COPY  E-VIEW-INVALIDATED
E-VIEW-TEMP  E-VISIBILITY  E-VOCAB-REMOVED
E-WIDEN-KIND  E-WIDEN-NARROW  E-WIDEN-SIGN
```

C 부록 C — 용어 색인 (Annex C: Index of terms)

1 이 문서에 나오는 용어를 가나다 차례로 모은다. 각 용어의 뜻은 적힌 조항에 있다.

> **참고 (informative)**
>
> 이 색인은 **생성된 것**이다. 사람이 손으로 고치지 아니한다.

표 75 — 용어 색인

용어	영어	조항
BOM	byte order mark	§3.7
access	access declaration	§6.4.3
effects	effect declaration	§6.4.3
ensures	postcondition	§6.4.3
errors	error condition	§6.4.3
fn	pure op	§6.4.1
mut_ref	exclusive reference	§8.4
op	op	§3.1
proc	effectful op	§6.4.1
ref	shared reference	§8.4
requires	precondition	§6.4.3
tests	examples	§6.4.3
가비지 컬렉터	garbage collector	§2.2
값	value	§3.2
개행	newline	§3.7
검사	check	§3.3
계약	contract	§1.3
구문	syntax	§1.1
구문 문법	syntactic grammar	§1.5
구현 정의	implementation-defined	§4.3
권한	capability	§3.4
규범	normative	§1.4
기계	target	§5.6
기본 연산	core op	§3.8
날 문자열	raw string	§6.1.4
낱말	keyword	§3.7
넓히기	widening	§3.8
능력	capability	§5.3
단락 평가	short-circuit	§3.8
단언	assertion	§6.4.13
닫개	closer	§3.7
덫 표현	trap representation	§3.2

덫 표현	trap representation	§8.7.1
등급	grade	§6.4.9
라이브러리 모듈	library module	§3.9
로우엔트	Lowent	§1.1
로컬 op	local op	§3.9
리터럴	literal	§3.2
리프	leaf	§9.3
모듈	module	§3.1
모호	open issue	§4.3
목적지	destination	§3.9
문맥	context	§3.9
문자 리터럴	character literal	§3.2
문자열 리터럴	string literal	§3.2
문장	statement	§3.8
미정의 동작	undefined behaviour	§3.3
번역	translation	§3.6
번역 단위	translation unit	§5.1
번역 환경	translation environment	§3.9
부동소수점 리터럴	floating-point literal	§3.2
빌드 모드	build mode	§3.3
빌트인 연산	builtin op	§3.8
사전 조건	precondition	§3.3
사후 조건	postcondition	§3.3
상표	brand	§3.2
생성 규칙	production	§1.5
선언	declaration	§3.1
섬	island	§3.7
소유	ownership	§3.5
순수	pure	§3.4
스스로를 품는 마디	self-owning node	§8.5
스테이지	stage	§3.9
슬라이스	slice	§3.2
시작점	entry point	§3.9
시험 블록	test block	§6.4.13
식	expression	§3.8
신뢰 경계	trust boundary	§3.3
실행 환경	execution environment	§3.9
안전한 부분집합	safe subset	§3.3

액터	actor	§3.8
어휘 문법	lexical grammar	§1.5
얼로케이터	allocator	§8.13
열거	enum	§3.2
영역	region	§3.5
옥텟	octet	§3.9
완결	completion	§8.5
의미 규칙	semantic rules	§1.1
의미 엔트로피	semantic entropy	§1.3
이름	identifier	§3.7
이스케이프	escape	§3.2
임시값	temporary	§3.9
자리	place	§3.9
자유 선택	free choice	§1.3
적합성	conformance	§1.1
적합한 처리기	conforming processor	§3.6
적합한 프로그램	conforming program	§3.6
전위 표기	prefix notation	§3.7
접두사	prefix	§3.2
정의	definition	§3.1
제약	constraints	§1.1
좁히기	narrowing	§3.8
종결자	terminal	§3.9
종료 상태	exit status	§3.9
주석	comment	§3.7
중위 표기	infix notation	§3.7
진단	diagnostic	§3.6
참고	informative	§1.4
참조	reference	§3.5
채널	channel	§10.6
처리기	processor	§1.1
최적화	optimisation	§3.9
타입	type	§3.2
탈출	escape	§3.5
텍스트 리터럴	text literal	§3.2
토큰	token	§3.7
투스 컴플리먼트	two’s complement	§3.9
트랩	trap	§3.3

트레이트	trait	§3.8
폭	width	§3.2
폼	form	§3.7
표지	marker	§6.2.15
표현	representation	§1.1
프로그램	program	§3.1
프리스탠딩	freestanding	§2.2
한정	qualification	§3.9
해제	release	§8.5
호스티드	hosted	§5.1
효과	effect	§3.4

D 부록 D — 기본 연산과 빌트인 연산 (Annex D: Core ops and builtin ops)

1 이 부록은 언어가 뜻을 정해 둔 연산을 모은다 — 이름과 한 줄짜리 뜻(D.1), 그리고 연산마다 **무엇을 받고 무엇을 내며 언제 거절하고 언제 멈추는지**(D.2)다.

> **참고 (informative)**
>
> 이 목록은 **생성된 것**이다. 사람이 손으로 고치지 아니한다 — 연산은 계속 늘고, 손으로 적은 목록은 곧 낡는다.

2 모두 234 개다 — 기본 연산 164 개와 빌트인 연산 70 개.

D.1 이름과 뜻

D.1.1 기본 연산 (core op)

2a 기본 연산은 어디서나 제 이름만으로 부른다. 저자가 지은 이름과 **한 이름공간**을 쓴다(§6.3.3).

표 76 — 기본 연산과 그 뜻

연산	무엇을 하는가
`abs`	절댓값. 가장 작은 음수에서는 넘치므로 트랩한다
`add`	두 수를 더한다. 넘치면 트랩한다
`all`	모두 참인가 — 첫 거짓에서 멈춘다
`alloc_bytes`	영역에서 바이트 자리를 얻는다
`and`	논리 곱 — 앞이 거짓이면 뒤를 **안 본다**
`any`	참이 하나라도 있나 — 첫 참에서 멈춘다
`atomic_add`	원자적으로 더한다
`atomic_and`	원자적 비트 곱
`atomic_cas`	기대한 값일 때만 바꾼다(compare-and-swap)
`atomic_fence`	메모리 순서의 울타리
`atomic_load`	원자적으로 읽는다
`atomic_or`	원자적 비트 합
`atomic_store`	원자적으로 쓴다
`atomic_sub`	원자적으로 뺀다
`atomic_swap`	원자적으로 바꿔 끼우고 옛 값을 낸다
`atomic_xor`	원자적 비트 배타합
`await`	그 흐름이 끝날 때까지 기다렸다가 결과를 낸다
`bit_and`	비트 곱
`bit_cast`	비트를 그대로 두고 타입만 바꾼다
`bit_not`	비트 뒤집기 — **폭 안에서** 뒤집는다(C 처럼 int 로 승격하지 아니한다)
`bit_or`	비트 합
`bit_xor`	비트 배타합
`bitset_complement`	여집합
`bitset_contains`	그 원소가 들어 있나

`bitset_count`	집합에 든 원소의 수
`bitset_difference`	차집합
`bitset_insert`	집합에 그 원소를 넣는다 — **제자리에서** 바꾼다
`bitset_intersect`	교집합
`bitset_is_empty`	비었는가
`bitset_is_subset`	앞이 뒤에 다 들어 있나
`bitset_new`	빈 비트 집합
`bitset_remove`	집합에서 그 원소를 지운다 — **제자리에서** 바꾼다
`bitset_union`	합집합
`borrow`	빌린 것의 수명을 스코프로 못 박는다
`byte_swap`	바이트 순서를 뒤집는다
`call_builtin`	계산 잎과 파일·그물 잎을 부르는 머리 — 다음 원자가 잎의 이름이다(닫힌 집합). 그 이름은 전역 어휘가 아니라 이 자리에만 산다. 낸 값은 그 잎이 내는 값
`capacity`	담을 수 있는 최대
`cast`	수치 변환(폭·부호·부동소수점)
`channel`	채널을 만든다
`chk_add`	더하기 — 넘쳤는지 **답과 함께** 돌려준다
`chk_mul`	곱하기 — 넘쳤는지 답과 함께 돌려준다
`chk_sub`	빼기 — 넘쳤는지 답과 함께 돌려준다
`chrecv`	채널에서 값을 뺀다
`chsend`	채널에 값을 넣는다
`collect`	파이프라인의 결과를 자리에 담는다
`config`	번역 시점의 설정 값
`copy`	주는 쪽의 원소를 받는 쪽에 베낀다
`count`	파이프라인의 원소 수를 센다(종결자)
`count_ones`	선 비트의 수
`cstr_of`	C 문자열로 본다
`deref`	참조가 가리키는 값
`div`	앞을 뒤로 나눈다. 0 으로 나누면 트랩한다
`div_nz`	0 이 아님이 증명된 나눗셈 — 검사 없이 나눈다
`drain`	그 액터의 우편함에 든 것을 처리한다
`elem_ge`	모든 원소가 그 값 이상이다
`elem_gt`	모든 원소가 그 값보다 크다
`elem_le`	모든 원소가 그 값 이하다
`elem_lt`	모든 원소가 그 값보다 작다
`encode`	주어진 부호로 바이트를 만든다
`enumerate`	원소에 차례 번호를 붙인다
`eq`	같은가

`error`	오류를 담는다
`error_value`	담긴 오류를 꺼낸다
`expect`	시험 블록 안의 단언 — 조건이 거짓이면 그 시험이 실패한다
`field`	값의 안을 읽는다
`filter`	술어가 참인 원소만 지나보낸다(파이프라인 스테이지)
`fold`	누산기에 차례로 접는다(파이프라인 종결자)
`ge`	앞이 크거나 같은가
`gt`	앞이 큰가
`idx`	그 자리의 원소
`include_text`	번역 때 그 경로(이 소스 파일의 폴더에서 본다)의 파일을 읽은 바이트 — 번역 때 도는 `fn` 의 입력 자리에서만 선다
`into`	같은 뜻의 다른 타입으로 옮긴다
`is_error`	오류인가
`is_none`	값이 없나
`is_ok`	성공인가
`is_some`	값이 있나
`isa`	지금 그 갈래인가를 묻는다
`lane_all`	가림막의 레인이 모두 켜졌나 (레인 판정 — pipe 종결자 `all` 과 다른 낱말)
`lane_any`	가림막의 레인이 하나라도 켜졌나 (레인 판정 — pipe 종결자 `any` 와 다른 낱말)
`lane_avg`	두 수의 평균 — 중간값을 넘침 없이 낸다
`lane_reverse`	차례를 뒤집는다
`lane_rotate`	주어진 방향으로 돌린다
`lane_select`	가림막에 따라 두 값 중 하나를 레인마다 고른다
`le`	앞이 작거나 같은가
`leading_zeros`	맨 앞의 0 비트 수
`len`	길이 — **원소의 수**다(바이트 수가 아니다)
`load`	그 자리에서 읽는다
`load_masked`	가려진 자리만 읽는다
`lt`	앞이 작은가
`map`	원소마다 op 을 적용한다(파이프라인 스테이지)
`max`	둘 중 큰 것
`min`	둘 중 작은 것
`mod`	나눈 나머지
`mul`	두 수를 곱한다. 넘치면 트랩한다
`mut_ref`	쓰기 참조를 만든다
`narrow`	더 좁은 폭으로. 안 들어가면 트랩한다
`narrow_sat`	좁히되 안 들어가면 끝값에서 멈춘다
`narrow_try`	좁히기 — 안 들어가면 `none` 을 낸다

`narrow_wrap`	좁히되 안 들어가면 감는다
`native_lanes`	이 기계가 한 번에 다루는 레인 수
`ne`	다른가
`neg`	부호를 뒤집는다
`nonzero_of`	0 이 아님을 증명해 담는다
`nonzero_or`	0 이면 0 아닌 대신을 낸다 — 결과는 0 이 아님이 보장된다
`not`	논리 부정
`ok`	성공한 답을 담는다
`ok_value`	담긴 성공값을 꺼낸다
`or`	논리 합 — 앞이 참이면 뒤를 안 본다
`panic`	계약이 깨졌음을 알리고 멈춘다
`payload`	갈래가 지닌 칸의 값을 꺼낸다(먼저 `isa` 로 그 갈래임을 확인해야 한다)
`pipe`	한 줄기로 흘린다 — **한 번의 훑기**다
`pop`	뒤에서 하나 뺀다
`prefetch`	곧 쓸 자리를 미리 끌어 온다
`push`	뒤에 붙인다
`range`	값의 범위를 좁힌 매개변수 표시
`read_volatile`	장치 레지스터를 읽는다 — 합치거나 재배치하지 아니한다
`reduce_add`	레인을 모두 더한다
`reduce_max`	레인 중 가장 큰 것
`reduce_min`	레인 중 가장 작은 것
`reduce_mul`	레인을 모두 곱한다
`ref`	읽기 참조를 만든다
`region`	영역을 연다 — 수명이 곧 스코프다
`ret`	오류를 위로 넘긴다
`rotl`	왼쪽으로 돌린다
`rotr`	오른쪽으로 돌린다
`same_slice`	두 슬라이스가 같은 바이트인가(시작과 길이)
`sat_add`	더하되 넘치면 **끝값에서 멈춘다**(포화)
`sat_mul`	곱하되 넘치면 끝값에서 멈춘다
`sat_sub`	빼되 넘치면 끝값에서 멈춘다
`scan`	접으며 중간값을 낸다
`send`	액터에 메시지를 보낸다
`shl`	왼쪽으로 민다. 시프트 양이 폭 이상이면 트랩한다
`shr`	오른쪽으로 민다. 부호 있는 값은 산술 이동이다
`shuffle`	레인을 주어진 차례로 다시 놓는다
`size_of`	그 타입이 차지하는 바이트 수
`skip`	앞의 n 개를 건너뛴다

`some`	값을 담은 option 을 만든다
`some_value`	담긴 값을 꺼낸다
`spawn`	새 실행 흐름을 만들고 핸들을 낸다
`splat`	한 값을 모든 레인에 채운다
`stack_count`	스택에 든 원소의 수
`stack_new`	스택 자료를 만든다
`store`	그 자리에 쓴다
`store_masked`	가려진 자리만 쓴다
`str_from_cstr`	C 문자열에서 읽는다
`sub`	앞에서 뒤를 뺀다. 넘치면 트랩한다
`subslice`	부분 슬라이스 — 복사하지 아니한다
`swap`	두 자리를 맞바꾼다
`take`	앞에서 n 개
`trailing_zeros`	맨 뒤의 0 비트 수
`try_view`	`view` 와 같되 계약이 깨지면 `none` 을 낸다
`unsafe_fn`	op 의 주소를 값으로 낸다(씨에 되부름으로 넘긴다)
`value_or`	값이 있으면 그것, 없으면 기본값 — 기본값은 **성공 시 평가되지 아니한다**
`view`	바이트를 구조체로 본다 — 복사 없음. 정렬·길이 계약을 확인한다
`view_array`	바이트를 그 타입의 배열로 본다 — 복사 없음
`view_segments`	조각 사슬을 하나의 배열처럼 본다
`widen`	더 넓은 폭으로 — 값이 보존된다
`wrap_add`	더하되 넘치면 **감는다**(모듈러)
`wrap_mul`	곱하되 넘치면 감는다
`wrap_shl`	왼쪽으로 밀되 시프트 양을 폭으로 감는다
`wrap_shr`	오른쪽으로 밀되 시프트 양을 폭으로 감는다
`wrap_sub`	빼되 넘치면 감는다
`write_volatile`	장치 레지스터에 쓴다 — 합치거나 재배치하지 아니한다
`zip`	두 줄기를 짝지어 흘린다

D.1.2 빌트인 연산 (builtin op)

2b 빌트인 연산은 `call_builtin <이름> …` 으로만 부른다. **제 이름공간을 따로** 가지므로, 저자는 같은 철자를 제 이름으로 지을 수 있다(§6.3.3). 시스템 라이브러리와 처리기를 짓는 데 쓰는 것이며, 응용 프로그램은 이것을 감싼 표준 라이브러리의 op 을 부른다(§9.3).

표 77 — 빌트인 연산과 그 뜻

연산	무엇을 하는가
`aes_ctr`	AES-128-CTR — 키 16 · 카운터 16 을 제자리에서 올린다. 낸 값은 흐른 바이트 수
`aes_gcm`	AES-128-GCM 한 덩이 — 흐름(aes_ctr)과 누산(ghash)을 한 바퀴에. 뜻은 그 둘을 차례로 부른 것과 같다. 낸 값은 흐른 바이트 수

`aes_round`	AES 한 라운드 — 상태 16 바이트를 제자리에서 (SubBytes·ShiftRows·MixColumns·AddRoundKey). 낸 값은 16
`aes_round_last`	마지막 AES 라운드 — MixColumns 없이. 낸 값은 16
`arg`	그 자리의 명령줄 인자
`ceil`	올림
`chacha20`	ChaCha20 (RFC 8439) — 키 32 · 카운터 블록 16(앞 4 = 블록 번호 리틀엔디언 · 뒤 12 = nonce)을 제자리에서 올린다. 낸 값은 흐른 바이트 수
`chacha_poly`	ChaCha20-Poly1305 한 덩이 — 흐름과 누산을 한 바퀴에. 뜻은 chacha20 을 돌리고 그 암호문을 poly1305 에 먹인 것과 같다. 낸 값은 흐른 바이트 수
`clmul_hi`	캐리 없는 곱셈의 **윗말** — 64×64 비트 곱의 위쪽 64 비트. 덧셈이 배타합인 셈이다
`clmul_lo`	캐리 없는 곱셈의 **아랫말** — 64×64 비트 곱의 아래쪽 64 비트. 기계에 명령이 있으면 그것으로, 없으면 같은 답을 내는 셈으로
`cos`	코사인
`crc32`	CRC-32 검사값
`dir_close`	디렉터리를 닫는다
`dir_make`	디렉터리를 만든다
`dir_open`	디렉터리를 연다
`dir_read`	다음 항목을 읽는다
`env_get`	환경 변수를 읽는다
`exp`	e 의 거듭제곱
`file_close`	파일을 닫는다
`file_open`	파일을 연다
`file_read`	파일에서 읽는다
`file_seek`	읽고 쓸 자리를 옮긴다
`file_type`	그 경로가 무엇인가(파일·디렉터리…)
`file_write`	파일에 쓴다
`floor`	내림
`fmod`	부동소수점 나머지
`ghash`	GHASH — GF(2^128) 누산. h 16 · z 16 을 제자리에서 고친다. 낸 값은 먹인 바이트 수
`hash_bytes`	바이트열의 해시
`link_type`	그 경로 자신이 무엇인가 — 심링크를 따라가지 아니한다
`log`	자연로그
`mul_hi`	곱의 **윗말** — 부호 없는 64×64 비트 곱의 위쪽 64 비트(아래쪽은 `wrap_mul`). 큰 수 산술의 가장 안쪽 셈이다
`net_accept`	들어온 연결을 받는다
`net_close`	닫는다
`net_connect`	주소와 포트로 연결을 건다
`net_listen`	듣는 소켓을 연다

net_pair	맞물린 소켓 한 쌍
net_port	그 듣는 소켓의 포트
net_recv	받는다
net_resolve	이름을 IPv4 주소로 바꾼다 — DNS. 첫 A 레코드 하나
net_send	보낸다
path_remove	지운다
path_rename	이름을 바꾼다
poly1305	Poly1305 (RFC 8439) — 상태(h 다섯 · r 다섯)를 제자리에서 누산한다. 열여섯의 배수가 아니면 꼬리를 0 으로 채운다. 낸 값은 먹인 바이트 수
pow	거듭제곱
proc_kill	띄운 프로그램을 강제로 끝낸다 — POSIX SIGKILL · Windows TerminateProcess. 손자는 끄지 않는다
proc_poll	안 막고 확인한다 — 끝났으면 종료 상태(낮은 32 비트 = 코드 · 비트 32 = 신호), 아직이면 UINT64_MAX
proc_read	띄운 프로그램의 출력(표준 출력 + 오류)을 안 막고 읽는다 — n 바이트 · 0 = 지금은 없다 · none = 끝
proc_spawn	NUL 로 이은 인자 목록으로 프로그램을 띄운다 — 표준 입력은 비고 출력은 파이프로 모인다
proc_wait	띄운 프로그램이 끝날 때까지 기다리고 핸들을 닫는다 — 종료 상태
r_read	reactor 로 읽는다
r_write	reactor 로 쓴다
random_bytes	OS 엔트로피로 버퍼를 채운다
reactor_new	reactor 를 만든다
read_in	표준 입력에서 읽는다
rng_next	난수를 낸다
round	반올림 — 0 에서 먼 쪽으로
sha256	SHA-256
sha384	SHA-384 — SHA-512 의 다른 시작값이고 앞 48 바이트다
sha512	SHA-512
sin	사인
sqrt	제곱근
sum_neumaier	부동 조각을 **보정하며** 더한다 — 오차가 항의 개수에 매이지 아니한다
sum_seq	부동 조각을 앞에서 뒤로 한 번 더한다 — 빠르고, 오차가 항의 개수에 비례한다
time_local	지금의 날짜와 시각(한 수에 묶어 낸다)
time_now	단조 시계의 지금(나노초)
time_sleep	주어진 밀리초만큼 잔다 — 실제로 잔 밀리초를 낸다
tty_raw	터미널의 날 입력 모드를 켜거나 끈다
tty_read	터미널에서 읽는다(지금 없으면 0)
tty_size	터미널의 행과 열(한 수에 묶어 낸다)

`write_out`	표준 출력에 쓴다

3 기본 연산의 이름이 모두 같은 규칙을 갖지는 아니한다. 대부분은 지역 이름으로 쓸 수 없으나 (§6.1.3), 일부는 **문맥 안에서만** 연산이므로 그 밖에서는 저자의 이름이 될 수 있다. 어느 쪽인지는 처리기가 정하며, 막히는 것은 진단으로 말한다.

참고 (informative)

실측: 기본 연산 164 개 가운데 지역 이름 선언이 막히는 것은 **141 개**, 쓸 수 있는 것은 **23 개**다 — `all` · `any` · `borrow` · `call_builtin` · `capacity` · `collect` · `copy` · `count` · `drain` · `enumerate` · `filter` · `fold` · `into` · `is_none` · `map` · `pipe` · `pop` · `region` · `ret` · `scan` · `skip` · `take` · `zip`. ★ 이 수를 여기 적는 까닭은, 하나로 뭉뚱그리면 **이름 충돌 규칙을 틀리게 말하기** 때문이다(RFC-0101 F-19). 뭉뚱그린 목록은 수가 맞아도 규칙이 틀린다.

문법 틀 (grammar shape) — 기본 연산 이름

`abs`	`add`	`aes_ctr`	`aes_gcm`
`aes_round`	`aes_round_last`	`all`	`alloc_bytes`
`and`	`any`	`arg`	`atomic_add`
`atomic_and`	`atomic_cas`	`atomic_fence`	`atomic_load`
`atomic_or`	`atomic_store`	`atomic_sub`	`atomic_swap`
`atomic_xor`	`await`	`bit_and`	`bit_cast`
`bit_not`	`bit_or`	`bit_xor`	`bitset_complement`
`bitset_contains`	`bitset_count`	`bitset_difference`	`bitset_insert`
`bitset_intersect`	`bitset_is_empty`	`bitset_is_subset`	`bitset_new`
`bitset_remove`	`bitset_union`	`borrow`	`byte_swap`
`call_builtin`	`capacity`	`cast`	`ceil`
`chacha20`	`chacha_poly`	`channel`	`chk_add`
`chk_mul`	`chk_sub`	`chrecv`	`chsend`
`clmul_hi`	`clmul_lo`	`collect`	`config`
`copy`	`cos`	`count`	`count_ones`
`crc32`	`cstr_of`	`deref`	`dir_close`
`dir_make`	`dir_open`	`dir_read`	`div`
`div_nz`	`drain`	`elem_ge`	`elem_gt`
`elem_le`	`elem_lt`	`encode`	`enumerate`
`env_get`	`eq`	`error`	`error_value`
`exp`	`expect`	`field`	`file_close`
`file_open`	`file_read`	`file_seek`	`file_type`
`file_write`	`filter`	`floor`	`fmod`
`fold`	`ge`	`ghash`	`gt`
`hash_bytes`	`idx`	`include_text`	`into`
`is_error`	`is_none`	`is_ok`	`is_some`
`isa`	`lane_all`	`lane_any`	`lane_avg`
`lane_reverse`	`lane_rotate`	`lane_select`	`le`
`leading_zeros`	`len`	`link_type`	`load`
`load_masked`	`log`	`lt`	`map`
`max`	`min`	`mod`	`mul`
`mul_hi`	`mut_ref`	`narrow`	`narrow_sat`
`narrow_try`	`narrow_wrap`	`native_lanes`	`ne`
`neg`	`net_accept`	`net_close`	`net_connect`
`net_listen`	`net_pair`	`net_port`	`net_recv`
`net_resolve`	`net_send`	`nonzero_of`	`nonzero_or`
`not`	`ok`	`ok_value`	`or`
`panic`	`path_remove`	`path_rename`	`payload`
`pipe`	`poly1305`	`pop`	`pow`
`prefetch`	`proc_kill`	`proc_poll`	`proc_read`
`proc_spawn`	`proc_wait`	`push`	`r_read`
`r_write`	`random_bytes`	`range`	`reactor_new`

```
read_in          read_volatile    reduce_add       reduce_max
reduce_min       reduce_mul       ref              region
ret              rng_next         rotl             rotr
round            same_slice       sat_add          sat_mul
sat_sub          scan             send             sha256
sha384           sha512           shl              shr
shuffle          sin              size_of          skip
some             some_value       spawn            splat
sqrt             stack_count      stack_new        store
store_masked     str_from_cstr    sub              subslice
sum_neumaier     sum_seq          swap             take
time_local       time_now         time_sleep       trailing_zeros
try_view         tty_raw          tty_read         tty_size
unsafe_fn        value_or         view             view_array
view_segments    widen            wrap_add         wrap_mul
wrap_shl         wrap_shr         wrap_sub         write_out
write_volatile   zip
```

D.2 받는 것과 내는 것

4 아래 표는 연산마다 **꼴**(적는 모양), **받는 것**(피연산자의 타입과 조건), **내는 것**(결과의 타입과 값), **번역 때 거절**(적합하지 아니한 프로그램에 내는 진단), **실행 때 멈춤**(트랩)을 적는다. – 는 «없다» 는 뜻이고, 빈칸은 **아직 재지 못했다**는 뜻이다 — 짐작으로 채우지 아니한다.

> **참고 (informative)**
>
> 이 절도 **생성된 것**이다. 줄마다 처리기를 돌려 잰 탐침이 뒷받침한다. 234 개 가운데 **102 개**를 적었고, 나머지는 D.1 의 한 줄짜리 뜻과 본문 조항이 말한다.

5 타입을 줄여 적는 말. **정수** 는 `u8` `u16` `u32` `u64` `i8` `i16` `i32` `i64`, **부동** 은 `f32` `f64`, **수** 는 정수 또는 부동이다. **맞는 둘** 은 넓히기로 한 타입에 맞출 수 있는 두 수다(§6.3.4) — 폭만 다르면 넓은 쪽으로, 부호가 섞이면 둘을 다 담는 부호 있는 쪽으로 맞춘다. `T` 는 그 자리에 적는 타입 이름이다. `usize` 와 `idiff` 는 이름이 다른 타입이라 위의 정수와 섞이지 아니한다.

표 78 — 산술

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`add a b` §6.3.4	**받는 것:** `a` `b` — 맞는 둘 **내는 것:** 둘 중 넓은 쪽 타입 **번역 때 거절:** 정수와 부동을 섞으면 `E-TYPE-MIX` · 부호가 섞여 값을 지키는 넓히기가 없으면 `E-TYPE-SIGN` · 리터럴이 타입에 안 들어가면 `E-TYPE-WIDTH` · 인자 수가 다르면 `E-IR-ARITY` **실행 때 멈춤:** 정수: 합이 그 폭에 안 들어가면 `E-VM-OVERFLOW`. 부동: 멈추지 않는다(무한 · NaN 이 값으로 나온다)
`sub a b` §6.3.4	**받는 것:** `a` `b` — 맞는 둘 **내는 것:** 둘 중 넓은 쪽 타입 **번역 때 거절:** 정수와 부동을 섞으면 `E-TYPE-MIX` · 부호가 섞여 값을 지키는 넓히기가 없으면 `E-TYPE-SIGN` · 리터럴이 타입에 안 들어가면 `E-TYPE-WIDTH` · 인자 수가 다르면 `E-IR-ARITY` **실행 때 멈춤:** 정수: 차가 그 폭에 안 들어가면 `E-VM-OVERFLOW`(부호 없는 타입에서 작은 수에서 큰 수를 빼는 것 포함). 부동: 멈추지 않는다

`mul a b` §6.3.4	**받는 것:** `a` `b` — 맞는 둘 **내는 것:** 둘 중 넓은 쪽 타입 **번역 때 거절:** 정수와 부동을 섞으면 `E-TYPE-MIX` · 부호가 섞여 값을 지키는 넓히기가 없으면 `E-TYPE-SIGN` · 리터럴이 타입에 안 들어가면 `E-TYPE-WIDTH` · 인자 수가 다르면 `E-IR-ARITY` **실행 때 멈춤:** 정수: 곱이 그 폭에 안 들어가면 `E-VM-OVERFLOW`. 부동: 멈추지 않는다
`div a b` §6.3.4	**받는 것:** `a` `b` — 맞는 둘 **내는 것:** 둘 중 넓은 쪽 타입. 정수의 몫은 0 쪽으로 자른다(`div -7 2` 는 −3) **번역 때 거절:** 정수와 부동을 섞으면 `E-TYPE-MIX` · 부호가 섞여 값을 지키는 넓히기가 없으면 `E-TYPE-SIGN` · 리터럴이 타입에 안 들어가면 `E-TYPE-WIDTH` · 인자 수가 다르면 `E-IR-ARITY` **실행 때 멈춤:** 정수: `b` 가 0 이면 `E-VM-DIV0` · 부호 있는 타입의 가장 작은 값을 −1 로 나누면 `E-VM-OVERFLOW`. 부동: 멈추지 않는다(0 으로 나누면 무한)
`mod a b` §6.3.4	**받는 것:** `a` `b` — 맞는 두 정수 **내는 것:** 둘 중 넓은 쪽 타입 — 바닥 나눗셈의 나머지. 부호는 `b` 를 따른다(`mod -7 2` 는 1, `mod 7 -2` 는 −1). `b` 가 양수이면 0 이상이고 `b` 보다 작다. 가장 작은 값을 −1 로 나누면 0 **번역 때 거절:** 정수와 부동을 섞으면 `E-TYPE-MIX` · 두 부동이면 `E-TYPE-KIND`(부동의 나머지는 `fmod` — 2026-11-02 부터 번역에서 거절한다) · 부호가 섞여 값을 지키는 넓히기가 없으면 `E-TYPE-SIGN` · 리터럴이 타입에 안 들어가면 `E-TYPE-WIDTH` · 인자 수가 다르면 `E-IR-ARITY` **실행 때 멈춤:** `b` 가 0 이면 `E-VM-DIV0`
`neg a` §6.3.3	**받는 것:** `a` — 부호 있는 정수 또는 부동 **내는 것:** `a` 의 타입 — 부호를 뒤집은 값 **번역 때 거절:** 부호 없는 정수를 주면 `E-TYPE-SIGN` · `E-IR-ARITY` **실행 때 멈춤:** 부호 있는 정수의 가장 작은 값이면 `E-VM-OVERFLOW`(뒤집은 값이 그 폭에 없다). 부동: 멈추지 않는다
`abs a` §6.3.9	**받는 것:** `a` — 부호 있는 정수 또는 부동 **내는 것:** `a` 의 타입 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** 부호 있는 정수의 가장 작은 값이면 `E-VM-OVERFLOW`(절댓값이 그 폭에 없다). 부동: 멈추지 않는다
`min a b` §6.3.3	**받는 것:** `a` `b` — 같은 부호의 두 정수, 또는 두 부동 **내는 것:** 둘 중 작은 값 — 둘 중 넓은 쪽 타입 **번역 때 거절:** 부호가 다른 두 정수면 `E-TYPE-SIGN` · `E-IR-ARITY` **실행 때 멈춤:** —
`max a b` §6.3.3	**받는 것:** `a` `b` — 같은 부호의 두 정수, 또는 두 부동 **내는 것:** 둘 중 큰 값 — 둘 중 넓은 쪽 타입 **번역 때 거절:** 부호가 다른 두 정수면 `E-TYPE-SIGN` · `E-IR-ARITY` **실행 때 멈춤:** —

표 79 — 넘침 처분

꼴	받는 것 · 내는 것 · 거절 · 멈춤

`wrap_add a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** 둘 중 넓은 쪽 타입 — `add` 의 결과를 폭 안으로 감은 값(2 의 거듭제곱으로 나눈 나머지) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`wrap_sub a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** 둘 중 넓은 쪽 타입 — `sub` 의 결과를 폭 안으로 감은 값(2 의 거듭제곱으로 나눈 나머지) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`wrap_mul a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** 둘 중 넓은 쪽 타입 — `mul` 의 결과를 폭 안으로 감은 값(2 의 거듭제곱으로 나눈 나머지) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`sat_add a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** 둘 중 넓은 쪽 타입 — `add` 의 결과를 폭의 끝값에 붙인 값(위로 넘치면 가장 큰 값, 아래로 넘치면 가장 작은 값) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`sat_sub a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** 둘 중 넓은 쪽 타입 — `sub` 의 결과를 폭의 끝값에 붙인 값(위로 넘치면 가장 큰 값, 아래로 넘치면 가장 작은 값) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`sat_mul a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** 둘 중 넓은 쪽 타입 — `mul` 의 결과를 폭의 끝값에 붙인 값(위로 넘치면 가장 큰 값, 아래로 넘치면 가장 작은 값) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`chk_add a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** `option T`(`T` 는 둘 중 넓은 쪽) — `add` 의 결과가 폭에 들어가면 그 값, 넘치면 없음 **번역 때 거절:** 부동을 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`chk_sub a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** `option T`(`T` 는 둘 중 넓은 쪽) — `sub` 의 결과가 폭에 들어가면 그 값, 넘치면 없음 **번역 때 거절:** 부동을 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`chk_mul a b` §6.3.6	**받는 것:** `a b` — 맞는 두 정수 **내는 것:** `option T`(`T` 는 둘 중 넓은 쪽) — `mul` 의 결과가 폭에 들어가면 그 값, 넘치면 없음

	번역 때 거절: 부동을 주면 E-TYPE-KIND · E-IR-ARITY **실행 때 멈춤:** —
nonzero_of a §6.3.6	**받는 것:** a — 정수 **내는 것:** option nonzero T(T 는 a 의 타입) — 0 이면 없음 **번역 때 거절:** 부동 · bool 을 주면 E-TYPE-KIND · E-IR-ARITY **실행 때 멈춤:** —
div_nz a b §6.3.6	**받는 것:** a — 정수 · b — nonzero_of 로 얻은 0 아닌 값 **내는 것:** a 의 타입 — 몫 **번역 때 거절:** b 가 0 아님이 보장되지 않으면 E-TYPE-NONZERO · E-IR-ARITY **실행 때 멈춤:** —
nonzero_or a d §6.3.6	**받는 것:** a — 정수 · d — a 와 같은 타입이고 0 아님이 보장된 값(0 아닌 리터럴 · nonzero_of 로 얻은 값 · 그것으로 묶은 let) **내는 것:** a 의 타입 — a 가 0 이 아니면 a, 0 이면 d. 0 아님이 보장되므로 div_nz 의 제수가 된다 **번역 때 거절:** d 가 0 아님이 보장되지 않으면 E-TYPE-NONZERO · 부동 · bool 은 E-TYPE-KIND · E-IR-ARITY **실행 때 멈춤:** —

표 80 — 비교

꼴	받는 것 · 내는 것 · 거절 · 멈춤
eq a b §6.3.3	**받는 것:** a b — 맞는 둘, 또는 두 bool **내는 것:** bool — 같으면 참. 부동의 NaN 은 자기 자신과도 같지 않다 **번역 때 거절:** 수와 bool 을 섞으면 E-TYPE-KIND · option · result 를 주면 E-TYPE-KIND · 부호가 섞이면 E-TYPE-SIGN · E-IR-ARITY **실행 때 멈춤:** —
ne a b §6.3.3	**받는 것:** a b — 맞는 둘, 또는 두 bool **내는 것:** bool — 다르면 참. 부동의 NaN 은 자기 자신과도 같지 않다 **번역 때 거절:** 수와 bool 을 섞으면 E-TYPE-KIND · option · result 를 주면 E-TYPE-KIND · 부호가 섞이면 E-TYPE-SIGN · E-IR-ARITY **실행 때 멈춤:** —
lt a b §6.3.3	**받는 것:** a b — 맞는 둘 **내는 것:** bool — a 가 작으면 참. 부동에서 한쪽이 NaN 이면 거짓 **번역 때 거절:** 정수와 부동을 섞으면 E-TYPE-MIX · 부호가 섞이면 E-TYPE-SIGN · E-IR-ARITY **실행 때 멈춤:** —
le a b §6.3.3	**받는 것:** a b — 맞는 둘 **내는 것:** bool — a 가 작거나 같으면 참. 부동에서 한쪽이 NaN 이면 거짓 **번역 때 거절:** 정수와 부동을 섞으면 E-TYPE-MIX · 부호가 섞이면 E-TYPE-SIGN · E-IR-ARITY **실행 때 멈춤:** —
gt a b §6.3.3	**받는 것:** a b — 맞는 둘 **내는 것:** bool — a 가 크면 참. 부동에서 한쪽이 NaN 이면 거짓 **번역 때 거절:** 정수와 부동을 섞으면 E-TYPE-MIX · 부호가 섞이면 E-TYPE-SIGN · E-IR-

	ARITY **실행 때 멈춤:** —
ge a b §6.3.3	**받는 것:** a b — 맞는 둘 **내는 것:** bool — a 가 크거나 같으면 참. 부동에서 한쪽이 NaN 이면 거짓 **번역 때 거절:** 정수와 부동을 섞으면 E-TYPE-MIX · 부호가 섞이면 E-TYPE-SIGN · E-IR-ARITY **실행 때 멈춤:** —

표 81 — 논리

꼴	받는 것 · 내는 것 · 거절 · 멈춤
and a b §6.3.3	**받는 것:** a b — bool **내는 것:** bool. a 가 거짓이면 b 를 계산하지 않는다 **번역 때 거절:** bool 이 아니면 E-TYPE-LOGICAL · E-IR-ARITY **실행 때 멈춤:** —
or a b §6.3.3	**받는 것:** a b — bool **내는 것:** bool. a 가 참이면 b 를 계산하지 않는다 **번역 때 거절:** bool 이 아니면 E-TYPE-LOGICAL · E-IR-ARITY **실행 때 멈춤:** —
not a §6.3.3	**받는 것:** a — bool **내는 것:** bool **번역 때 거절:** bool 이 아니면 E-TYPE-LOGICAL · E-IR-ARITY **실행 때 멈춤:** —

표 82 — 비트

꼴	받는 것 · 내는 것 · 거절 · 멈춤
bit_and a b §6.3.5	**받는 것:** a b — 맞는 두 정수 **내는 것:** 두 피연산자가 같은 타입이면 그 타입 · 한쪽이 리터럴이면 다른 쪽의 타입 · 폭이나 부호만 다르면 산술과 같다 — 넓히기로 비교할 수 있으면 넓은 쪽 타입 **번역 때 거절:** E-IR-ARITY · usize · idiff 와 고정 폭을 섞으면 E-TYPE-NOMINAL · 넓히기로 비교할 수 없으면 E-TYPE-SIGN **실행 때 멈춤:** 부동을 주면 E-BITOP-TYPE
bit_or a b §6.3.5	**받는 것:** a b — 맞는 두 정수 **내는 것:** 두 피연산자가 같은 타입이면 그 타입 · 한쪽이 리터럴이면 다른 쪽의 타입 · 폭이나 부호만 다르면 산술과 같다 — 넓히기로 비교할 수 있으면 넓은 쪽 타입 **번역 때 거절:** E-IR-ARITY · usize · idiff 와 고정 폭을 섞으면 E-TYPE-NOMINAL · 넓히기로 비교할 수 없으면 E-TYPE-SIGN **실행 때 멈춤:** 부동을 주면 E-BITOP-TYPE
bit_xor a b §6.3.5	**받는 것:** a b — 맞는 두 정수 **내는 것:** 두 피연산자가 같은 타입이면 그 타입 · 한쪽이 리터럴이면 다른 쪽의 타입 · 폭이나 부호만 다르면 산술과 같다 — 넓히기로 비교할 수 있으면 넓은 쪽 타입 **번역 때 거절:** E-IR-ARITY · usize · idiff 와 고정 폭을 섞으면 E-TYPE-

	NOMINAL · 넓히기로 비교할 수 없으면 E-TYPE-SIGN **실행 때 멈춤:** 부동을 주면 E-BITOP-TYPE
bit_not a §6.3.5	**받는 것:** a — 정수 **내는 것:** a 의 타입 — 그 폭 안에서 뒤집는다(bit_not 을 u8 0 에 대면 255) **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** 부동을 주면 E-BITOP-TYPE
shl a n §6.3.5	**받는 것:** a — 정수 · n — 부호 없는 정수(부호 있는 타입은 E-SHIFT-SIGNED), a 의 폭보다 작다 **내는 것:** a 의 타입 — 왼쪽으로 옮긴다. 폭 밖으로 밀려난 비트는 버린다(넘침으로 치지 않는다) **번역 때 거절:** n 이 폭 이상임이 번역 때 보이면 E-SHIFT-RANGE · E-IR-ARITY **실행 때 멈춤:** n 이 음수이거나 폭 이상이면 E-VM-SHIFT · 부동을 주면 E-BITOP-TYPE
shr a n §6.3.5	**받는 것:** a — 정수 · n — 부호 없는 정수(부호 있는 타입은 E-SHIFT-SIGNED), a 의 폭보다 작다 **내는 것:** a 의 타입 — 오른쪽으로 옮긴다. 부호 없는 타입은 0 이, 부호 있는 타입은 부호 비트가 들어온다 **번역 때 거절:** n 이 폭 이상임이 번역 때 보이면 E-SHIFT-RANGE · E-IR-ARITY **실행 때 멈춤:** n 이 음수이거나 폭 이상이면 E-VM-SHIFT · 부동을 주면 E-BITOP-TYPE
wrap_shl a n §6.3.5	**받는 것:** a — 정수 · n — 부호 없는 정수(부호 있는 타입은 E-SHIFT-SIGNED) **내는 것:** a 의 타입 — n 을 폭으로 나눈 나머지만큼 왼쪽으로 옮긴다(폭 8 에서 9 칸은 1 칸) **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
wrap_shr a n §6.3.5	**받는 것:** a — 정수 · n — 부호 없는 정수(부호 있는 타입은 E-SHIFT-SIGNED) **내는 것:** a 의 타입 — n 을 폭으로 나눈 나머지만큼 오른쪽으로 옮긴다(폭 8 에서 9 칸은 1 칸) **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
rotl a n §6.3.5	**받는 것:** a — 정수 · n — 정수 **내는 것:** a 의 타입 — 왼쪽으로 돌린다. 밀려난 비트가 반대쪽으로 들어온다. n 은 폭으로 나눈 나머지로 읽는다 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** 부동을 주면 E-BITOP-TYPE
rotr a n §6.3.5	**받는 것:** a — 정수 · n — 정수 **내는 것:** a 의 타입 — 오른쪽으로 돌린다. 밀려난 비트가 반대쪽으로 들어온다. n 은 폭으로 나눈 나머지로 읽는다

	번역 때 거절: `E-IR-ARITY` **실행 때 멈춤:** 부동을 주면 `E-BITOP-TYPE`
`count_ones a` §6.3.5	**받는 것:** `a` — 정수 **내는 것:** `a` 의 타입 — 1 인 비트의 수. 0 이상 폭 이하 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`leading_zeros a` §6.3.5	**받는 것:** `a` — 정수 **내는 것:** `a` 의 타입 — 맨 윗자리부터 이어지는 0 의 수. `a` 가 0 이면 폭(`u8` 에서 8) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** 부동을 주면 `E-BITOP-TYPE`
`trailing_zeros a` §6.3.5	**받는 것:** `a` — 정수 **내는 것:** `a` 의 타입 — 맨 아랫자리부터 이어지는 0 의 수. `a` 가 0 이면 폭 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** 부동을 주면 `E-BITOP-TYPE`
`byte_swap a` §6.3.5	**받는 것:** `a` — 정수 **내는 것:** `a` 의 타입 — 바이트 차례를 뒤집은 값. 한 바이트 타입에서는 그대로 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** 부동을 주면 `E-BITOP-TYPE`
`call_builtin clmul_lo a b` §6.3.3	**받는 것:** `a b` — `u64` **내는 것:** `u64` — 캐리 없는 64×64 비트 곱의 아래쪽 64 비트 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin clmul_hi a b` §6.3.3	**받는 것:** `a b` — `u64` **내는 것:** `u64` — 캐리 없는 64×64 비트 곱의 위쪽 64 비트 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin mul_hi a b` §6.3.3	**받는 것:** `a b` — `u64` **내는 것:** `u64` — 64×64 비트 곱(부호 없음)의 위쪽 64 비트. 아래쪽은 `wrap_mul` **번역 때 거절:** `E-IR-ARITY` · `E-TYPE-ARG`(`u64` 가 아닌 입력) **실행 때 멈춤:** —

표 83 — 읽기

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`len s` §6.2.6	**받는 것:** `s` — 슬라이스 · 배열 · 문자열 리터럴 · 권한 `cap args`(받은 인자의 수 — 진입 `proc` 만 그 권한을 받으므로 `fn` 탐침은 없다) **내는 것:** `usize` — 원소의 수(바이트 수가 아니다) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`idx s i` §6.2.6	**받는 것:** `s` — 슬라이스 · 배열 · `i` — `usize` **내는 것:** 원소의 타입 — `i` 번째 원소

	번역 때 거절: `i` 가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** `i` 가 길이 밖이면 `E-VM-BOUNDS`
`subslice s a b` §6.2.6	**받는 것:** `s` — 슬라이스 · `a b` — `usize` **내는 것:** `slice T`(`T` 는 `s` 의 원소) — `[a, b)` 를 복사 없이 보는 슬라이스 **번역 때 거절:** 경계가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** `a` 가 `b` 보다 크거나 `b` 가 길이 밖이면 `E-VM-BOUNDS`
`view_array T s` §6.2.13	**받는 것:** `T` — 정수 또는 부동 타입 · `s` — 바이트 슬라이스 **내는 것:** `slice T` — 같은 바이트를 `T` 의 나열로 본다(복사 없음) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** `s` 의 길이가 `T` 의 크기의 배수가 아니면 `E-VM-VIEW`

표 84 — 집합

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`bitset_count s` §6.2.12	**받는 것:** `s` — `bitset n` **내는 것:** `usize` — 든 원소의 수(0 이상 `n` 이하) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —

표 85 — 부동소수점 함수

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`call_builtin sqrt a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입 — 제곱근. 음수면 NaN **번역 때 거절:** 정수를 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin sin a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입 — 사인(라디안) **번역 때 거절:** 정수를 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin cos a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입 — 코사인(라디안) **번역 때 거절:** 정수를 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin exp a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입 — e 의 거듭제곱 **번역 때 거절:** 정수를 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin log a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입 — 자연로그. 0 이면 음의 무한, 음수면 NaN **번역 때 거절:** 정수를 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin floor a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입(부동 그대로다 — 정수 타입으로 바꾸려면 `cast`) — `a` 보다 크지 않은 가장 큰 정수 값(`floor -1.5` 는 −2.0)

	번역 때 거절: `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin ceil a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입(부동 그대로다 — 정수 타입으로 바꾸려면 `cast`) — `a` 보다 작지 않은 가장 작은 정수 값 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin round a` §6.3.9	**받는 것:** `a` — 부동 **내는 것:** `a` 의 타입(부동 그대로다 — 정수 타입으로 바꾸려면 `cast`) — 가장 가까운 정수 값. 꼭 가운데면 0 에서 먼 쪽(`round 2.5` 는 3.0, `round -2.5` 는 −3.0) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin pow a b` §6.3.9	**받는 것:** `a` `b` — 같은 부동 타입 **내는 것:** `a` 의 타입 — `a` 의 `b` 제곱. 음수 밑에 정수 아닌 지수면 NaN **번역 때 거절:** 정수를 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin fmod a b` §6.3.9	**받는 것:** `a` `b` — 같은 부동 타입 **내는 것:** `a` 의 타입 — 나머지. 부호는 `a` 를 따른다(`fmod -7.5 2.0` 은 −1.5) **번역 때 거절:** 정수를 주면 `E-TYPE-KIND` · `E-IR-ARITY` **실행 때 멈춤:** —

표 86 — 변환

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`widen T a` §6.2.5	**받는 것:** `T` — 정수 또는 부동 타입 · `a` — 같은 갈래이고 `T` 보다 넓지 않은 수 **내는 것:** `T` — 값은 그대로다 **번역 때 거절:** `T` 가 더 좁으면 `E-WIDEN-NARROW` · 부호 있는 값을 부호 없는 `T` 로 넓히면 `E-WIDEN-SIGN` · 정수와 부동을 건너면 `E-WIDEN-KIND` · `T` 를 안 적으면 `E-FORM-UNEXPECTED` · `a` 가 참거짓이면 `E-WIDEN-KIND`(`cast` 로 적는다) · `usize` · `idiff` 가 원천이나 `T` 이면 `E-TYPE-NOMINAL`(`cast` 로 적는다 — 폭이 구현에 달려 있다) **실행 때 멈춤:** —
`narrow T a` §6.3.6	**받는 것:** `T` — 정수 또는 부동 타입 · `a` — `T` 와 같은 갈래의 수 **내는 것:** `T` — 값은 그대로다 **번역 때 거절:** `T` 를 안 적으면 `E-FORM-UNEXPECTED` · `usize` · `idiff` 가 원천이나 `T` 이면 `E-TYPE-NOMINAL`(`cast` 로 적는다) **실행 때 멈춤:** 정수 값이 `T` 에 안 들어가면 `E-VM-CAST`
`narrow_wrap T a` §6.3.6	**받는 것:** `T` — 정수 타입 · `a` — 수 **내는 것:** `T` — 아랫비트만 남긴 값(`narrow_wrap u8` 에 256 은 0, −1 은 255) **번역 때 거절:** `T` 를 안 적으면 `E-FORM-UNEXPECTED` **실행 때 멈춤:** —
`narrow_sat T a` §6.3.6	**받는 것:** `T` — 정수 타입 · `a` — 수 **내는 것:** `T` — `T` 의 끝값에 붙인 값(`narrow_sat u8` 에 300 은 255, −1 은 0)

	번역 때 거절: `T` 를 안 적으면 `E-FORM-UNEXPECTED` **실행 때 멈춤:** —
`narrow_try T a` §6.3.6	**받는 것:** `T` — 정수 타입 · `a` — 정수 **내는 것:** `option T` — 값이 `T` 에 들어가면 그 값, 아니면 없음 **번역 때 거절:** `T` 를 안 적으면 `E-FORM-UNEXPECTED` **실행 때 멈춤:** —
`cast T a` §6.2.16	**받는 것:** `T` — 정수 또는 부동 타입 · `a` — 수, 또는 참거짓(`T` 가 정수 타입일 때) **내는 것:** `T`. 부동에서 정수로 가면 소수부를 버린다(`cast i8` 에 −1.9 는 −1). 정수에서 부동으로 가면 가장 가까운 값. 참거짓에서 오면 참은 1, 거짓은 0 **번역 때 거절:** `T` 가 `bool` 이면 `E-TYPE-KIND`(수를 참거짓으로 바꾸는 `cast` 는 없다 — `ne a. 0 .`) · 참거짓을 부동 타입으로 바꾸면 `E-TYPE-KIND` **실행 때 멈춤:** 정수 값이 정수 `T` 에 안 들어가면 `E-VM-CAST`
`bit_cast T a` §6.2.5	**받는 것:** `T` — 정수 또는 부동 타입 · `a` — 폭이 같은 수 **내는 것:** `T` — 비트는 그대로 두고 타입만 바꾼다(`bit_cast u64` 에 `f64` 1.0 은 4607182418800017408) **번역 때 거절:** **실행 때 멈춤:** —
`size_of T` §6.3.8	**받는 것:** `T` — 타입 **내는 것:** `usize` — 그 타입이 차지하는 바이트 수(`size_of u32` 는 4) **번역 때 거절:** **실행 때 멈춤:** —
`include_text "<경로>"` §6.8.1	**받는 것:** `"<경로>"` — 따옴표 문자열. 이 소스 파일의 폴더에서 본다(절대 경로 · `..` 은 안 된다) **내는 것:** `slice u8` — 번역 때 읽은 그 파일의 바이트(읽기 전용). 번역 때 도는 `fn` 의 입력 자리에서만 선다 **번역 때 거절:** 다른 자리면 `E-INCLUDE-PLACE` · 읽을 수 없거나 폴더 밖이거나 1 MiB 를 넘으면 그 `comptime` 자리가 그 까닭을 말하며 거절된다 (§6.8.1 (1g)) **실행 때 멈춤:** —

표 87 — 해시

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`call_builtin hash_bytes s` §6.3.3	**받는 것:** `s` — `slice u8` **내는 것:** `u64` — 바이트열의 해시 값(FNV-1a 64). 길이가 아니다 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin crc32 s` §6.3.3	**받는 것:** `s` — `slice u8` **내는 것:** `u32` — 반사 CRC-32(다항식 0xEDB88320) 검사값 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin sha256 s o` §6.3.3	**받는 것:** `s` — `slice u8` · `o` — `mut slice u8`(32 바이트 이상) **내는 것:** `usize` — `o` 에 써 넣은 바이트 수(32). `o` 가 짧으면 아무것도

	쓰지 않고 0 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
call_builtin sha384 s o §6.3.3	**받는 것:** s — slice u8 · o — mut slice u8(48 바이트 이상) **내는 것:** usize — o 에 써 넣은 바이트 수(48). o 가 짧으면 아무것도 쓰지 않고 0 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
call_builtin sha512 s o §6.3.3	**받는 것:** s — slice u8 · o — mut slice u8(64 바이트 이상) **내는 것:** usize — o 에 써 넣은 바이트 수(64). o 가 짧으면 아무것도 쓰지 않고 0 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —

표 88 — 암호

꼴	받는 것 · 내는 것 · 거절 · 멈춤
call_builtin aes_round st rk §6.3.3	**받는 것:** st — mut slice u8(16 바이트 상태) · rk — slice u8(16 바이트 라운드 키) **내는 것:** usize — 고친 상태의 바이트 수(16). SubBytes · ShiftRows · MixColumns · AddRoundKey 를 st 에 제자리에서 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
call_builtin aes_round_last st rk §6.3.3	**받는 것:** st — mut slice u8(16) · rk — slice u8(16) **내는 것:** usize — 고친 상태의 바이트 수(16). MixColumns 없이 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
call_builtin aes_ctr k c s d §6.3.3	**받는 것:** k — slice u8(16) · c — mut slice u8(카운터 16 · 제자리에서 올린다) · s — slice u8 · d — mut slice u8 **내는 것:** usize — 흐른 바이트 수(s 의 길이를 넘지 않는다) **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
call_builtin ghash h z data §6.3.3	**받는 것:** h — slice u8(16) · z — mut slice u8(누산 16 · 제자리에서) · data — slice u8 **내는 것:** usize — 먹인 바이트 수 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —
call_builtin chacha20 k c s d §6.3.3	**받는 것:** k — slice u8(32) · c — mut slice u8(카운터 블록 16 · 제자리에서 올린다) · s — slice u8 · d — mut slice u8 **내는 것:** usize — 흐른 바이트 수 **번역 때 거절:** E-IR-ARITY **실행 때 멈춤:** —

`call_builtin poly1305 st data` §6.3.3	**받는 것:** `st` — `mut slice`(원소 열 이상 · 제자리에서 누산) · `data` — `slice u8` **내는 것:** `usize` — 먹인 바이트 수 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin aes_gcm k c h z s d` §6.3.3	**받는 것:** `k` — `slice u8`(16) · `c` — `mut slice u8`(카운터 16) · `h` — `slice u8`(16) · `z` — `mut slice u8`(누산 16) · `s` — `slice u8` · `d` — `mut slice u8` **내는 것:** `usize` — 흐른 바이트 수. 뜻은 `aes_ctr` 뒤에 `ghash`를 부른 것과 같다 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`call_builtin chacha_poly k c st s d` §6.3.3	**받는 것:** `k` — `slice u8`(32) · `c` — `mut slice u8`(카운터 블록 16) · `st` — `mut slice u64`(원소 열 이상) · `s` — `slice u8` · `d` — `mut slice u8` **내는 것:** `usize` — 흐른 바이트 수. 뜻은 `chacha20` 뒤에 그 암호문을 `poly1305` 에 먹인 것과 같다 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —

표 89 — 난수

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`call_builtin rng_next s` §6.3.3	**받는 것:** `s` — `u64`(상태 — 씨는 부르는 쪽이 든다) **내는 것:** `u64` — 다음 값. 같은 상태에는 늘 같은 값 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —

표 90 — 원자

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`atomic_load s i [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` **내는 것:** 원소의 타입 — 그 칸의 값 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`
`atomic_add s i v [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` · `v` — 원소의 타입 **내는 것:** 원소의 타입 — 바꾸기 전의 값. 칸에는 합이 남는다 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`
`atomic_sub s i v [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` · `v` — 원소의 타입 **내는 것:** 원소의 타입 — 바꾸기 전의 값. 칸에는 차가 남는다 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`

`atomic_and s i v [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` · `v` — 원소의 타입 **내는 것:** 원소의 타입 — 바꾸기 전의 값. 칸에는 비트 곱이 남는다 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`
`atomic_or s i v [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` · `v` — 원소의 타입 **내는 것:** 원소의 타입 — 바꾸기 전의 값. 칸에는 비트 합이 남는다 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`
`atomic_xor s i v [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` · `v` — 원소의 타입 **내는 것:** 원소의 타입 — 바꾸기 전의 값. 칸에는 배타적 비트 합이 남는다 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`
`atomic_swap s i v [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` · `v` — 원소의 타입 **내는 것:** 원소의 타입 — 바꾸기 전의 값. 칸에는 `v` 가 남는다 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`
`atomic_cas s i want v [order o]` §7.5	**받는 것:** `s` — `mut slice` 정수 · `i` — `usize` · `want` `v` — 원소의 타입 **내는 것:** 원소의 타입 — 칸이 `want` 였으면 `v` 로 바꾸고 1, 아니면 그대로 두고 0 **번역 때 거절:** 첨자가 `usize` 가 아니면 `E-TYPE-NOMINAL` · `E-IR-ARITY` **실행 때 멈춤:** 첨자가 길이 밖이면 `E-VM-BOUNDS`

표 91 — 감싸개

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`some a` §6.2	**받는 것:** `a` — 값 **내는 것:** `option T`(`T` 는 `a` 의 타입) — 있음 **번역 때 거절:** 알맹이 타입이 다른 `option` 자리에 넣으면 `E-TYPE-WIDTH` · `E-IR-ARITY` **실행 때 멈춤:** —
`some_value o` §6.2	**받는 것:** `o` — `option T` **내는 것:** `T` — 알맹이 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** `o` 가 없음이면 `E-VM-NONE`
`is_some o` §6.2	**받는 것:** `o` — `option T` **내는 것:** `bool` — 있음이면 참 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —

`is_none o` §6.2	**받는 것:** `o` — `option T` **내는 것:** `bool` — 없음이면 참 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`value_or o d` §6.2	**받는 것:** `o` — `option T` · `d` — `T` **내는 것:** `T` — 있음이면 알맹이, 없음이면 `d` **번역 때 거절:** `d` 가 `T` 에 들어가지 않으면 `E-TYPE-WIDTH` · `E-IR-ARITY` **실행 때 멈춤:** —

표 92 — SIMD

꼴	받는 것 · 내는 것 · 거절 · 멈춤
`splat a` §6.3.11	**받는 것:** `a` — 정수 또는 부동 **내는 것:** `vec N T` — 모든 레인이 `a. T` 와 `N` 은 받는 자리(`vec N T`)가 정한다 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`reduce_add v` §6.3.11	**받는 것:** `v` — `vec N T` **내는 것:** `T` — 레인의 합(부호 있는 레인은 부호로 더한다) **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** 합이 `T` 에 안 들어가면 `E-VM-OVERFLOW`
`reduce_max v` §6.3.11	**받는 것:** `v` — `vec N T` **내는 것:** `T` — 가장 큰 레인 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —
`reduce_min v` §6.3.11	**받는 것:** `v` — `vec N T` **내는 것:** `T` — 가장 작은 레인 **번역 때 거절:** `E-IR-ARITY` **실행 때 멈춤:** —

E 부록 E — 모호 목록 (Annex E: Open issues)

1 이 부록은 **모호**(open issue)의 자리를 모은다(§4.3) — 이 문서가 아직 정하지 않았고 나중에 정해야 하는 자리다. 그런 자리에서 이 문서는 하나를 고르지 아니하고, 본문에 «모호» 상자로 표시한 뒤 가능한 풀이를 나란히 적는다.

2 적합한 프로그램은 이 목록의 **열린** 자리에 기대지 아니한다(§4.1). «처리기가 지금 하는 것» 은 잰 사실이지 규범이 아니다 — 풀이가 정해지면 달라질 수 있다. «권고» 도 규범이 아니다.

3 모호는 **자유 선택 (free choice)** 과 다르다(부록 F). 모호는 **정해야 하는데 아직 정하지 않은 것**이고, 뒤엣것은 **열어 두기로 정한 것**이다.

> **참고 (informative)**
>
> 이 목록은 **생성된 것**이다(`docs/spec/AMBIGUITY.tsv`). 번호는 다시 쓰지 아니한다 — 정해진 줄도 남는다.

4 모두 8 줄이고 그 가운데 열린 것이 1 줄이다.

표 93 — 모호 목록

번호	조항	물음 · 풀이 · 처리기가 지금 하는 것 · 권고 · 상태
M-0001	§6.3.4	**`div` 와 `mod` 의 음수 나머지** 풀이 ㄱ: 몫은 0 쪽으로 자르고 나머지는 피제수의 부호를 따른다 풀이 ㄴ: 몫은 아래로 내리고 나머지는 제수의 부호를 따른다 **처리기가 지금 하는 것:** 둘을 섞는다: `div -7 2` 는 −3, `mod -7 2` 는 1, `mod 7 -2` 는 −1 **권고:** (철회) 처음에 풀이 ㄱ 을 권했으나 틀린 권고였다 — 이 자리는 이미 정해져 있었다 **상태:** 정함: 2026-10-08 — 열린 것이 아니었다. 2026-08 의 결정이 이미 정했고 이 문서에 문장이 빠져 있었다: `div` 는 자르고 `mod` 는 제수의 부호를 따른다(§6.3.4 (5a) (5c))
M-0002	§6.3.4	**가장 작은 값을 −1 로 나눈 나머지** 풀이 ㄱ: `div` 와 같이 멈춘다 풀이 ㄴ: 0 을 낸다 **처리기가 지금 하는 것:** 0 을 낸다 **권고:** 풀이 ㄴ — 나머지는 0 으로 그 타입 안에 있다. 멈출 까닭이 없다 **상태:** 정함: 2026-10-08 풀이 ㄴ — 0 을 낸다(§6.3.4 (5d))
M-0003	§6.3.3	**`min` 과 `max` 에 NaN 이 끼면** 풀이 ㄱ: NaN 이 아닌 쪽을 낸다 풀이 ㄴ: NaN 을 낸다 **처리기가 지금 하는 것:** 인자의 차례에 따라 다르다: `min <NaN> 1.0` 은 1.0, `min 1.0 <NaN>` 은 NaN **권고:** 자유 선택으로 둔다 **상태:** 정함: 2026-10-08 자유 선택 FC-0002 로 옮김
M-0004	§6.3.3	**`neg` 를 쓸 수 있는 타입과 가장 작은 값** 풀이 ㄱ: 부호 없는 정수는 거부하고 가장 작은 값에서는 멈춘다 풀이 ㄴ: 폭 안으로 감은 값을 낸다 **처리기가 지금 하는 것:** (정해지기 전) 둘 다 받고 그 타입에 없는 값을 냈다(`i8` 의 −128 ⟶ 128) **권고:** 풀이 ㄱ — 산술은 결과가 폭에 안 들어가면 멈춘다(§6.3.4). `abs` 가 같은 자리에서 이미 멈춘다

		상태: 정함: 2026-10-08 풀이 ㄱ — 2026-08 의 결정이 이미 정했고 처리기가 따르지 않던 것이다. 부호 없는 정수는 `E-TYPE-SIGN`, 넘치면 멈춘다(§6.3.3 (8)(8a))
M-0005	§6.5.3.1	**`mut` 없는 `for` 원소에 `set`** 풀이 ㄱ: 번역이 거부된다 풀이 ㄴ: 그 바퀴의 사본만 바꾼다 **처리기가 지금 하는 것:** (정해지기 전) 받아들이고 슬라이스는 바꾸지 않았다 **권고:** 풀이 ㄱ — 아무것도 바꾸지 못하는 `set` 은 읽는 사람을 속인다. 바꾸려면 `mut` 을 적는다 **상태:** 정함: 2026-10-08 풀이 ㄱ — `E-FOR-HEAD` (§6.5.3.1 (3a))
M-0006	§6.1.1	**`note` 를 여는 줄에 태그가 없으면** 풀이 ㄱ: 번역이 거부된다 풀이 ㄴ: 그 줄만 주석이다 **처리기가 지금 하는 것:** (정해지기 전) 그 줄만 건너뛰었다 **권고:** 풀이 ㄱ — 줄 주석은 `rem` 이 한다. 한 가지 일에 두 철자를 두지 않는다 **상태:** 정함: 2026-10-08 풀이 ㄱ — `E-NOTE-TERM` (§6.1.1 (2a))
M-0007	§6.3.9	**초월 함수 결과의 마지막 비트** 풀이 ㄱ: 자유 선택으로 두고 범위를 적는다 풀이 ㄴ: 구현 정의로 두고 처리기가 셈 라이브러리를 적는다 풀이 ㄷ: 정확히 반올림한 값으로 정한다 **처리기가 지금 하는 것:** 기계의 셈 라이브러리가 내는 값을 그대로 낸다 **권고:** 풀이 ㄱ — 빠른 구현을 고를 자리다. 다만 범위(참값에서 얼마나 벗어나도 되는가)를 정해야 한다 **상태:** 열림
M-0008	§6.1.2	**예약만 해 둔 낱말이 있는가** 풀이 ㄱ: 없다 — 낱말 마흔넷이 모두 어떤 구문에 쓰인다 풀이 ㄴ: 있다 — 어느 것인지 적어야 한다 **처리기가 지금 하는 것:** 낱말 마흔넷이 모두 저장소의 코드에서 쓰인다(재어 확인) **권고:** 풀이 ㄱ **상태:** 정함: 2026-10-08 풀이 ㄱ — 재어 보니 예약만 된 낱말이 없다. 본문의 옛 문장이 낡은 것이었다(§6.1.2 (2a))

F 부록 F — 자유 선택 목록 (Annex F: Free choices)

1 이 부록은 **자유 선택 (free choice)** 의 자리를 모은다(§4.3). 그것은 이 문서가 허용하는 둘 이상의 동작이나 결과 가운데 하나를 고르는 것이며, 처리기는 어느 것을 골랐는지 문서로 적을 의무가 없다.

2 **이 목록이 전부다.** 여기 없는 자리는 자유 선택이 아니다. 처리기는 «고를 수 있는 범위» 안에서만 고르고, «정해져 있는 것» 은 어느 처리기에서나 같다. 적합한 프로그램은 처리기가 어느 것을 고르는가에 기대지 아니한다.

> **참고 (informative)**
>
> 이 목록은 **생성된 것**이다(`docs/spec/FREE-CHOICE.tsv`). 번호는 다시 쓰지 아니한다.

3 모두 4 줄이다.

표 94 — 자유 선택 목록

번호	조항	자리 · 고를 수 있는 범위 · 정해져 있는 것
FC-0001	§6.3.1	**피연산자를 계산하는 차례** **고를 수 있는 범위:** 한 연산의 피연산자들 가운데 효과를 내지 않는 것들의 모든 차례 **정해져 있는 것:** 모두 계산한 뒤에 연산한다 · 피연산자마다 한 번씩만 계산한다 · 효과(`panic` 밖의 것)를 내는 피연산자들은 적힌 차례로 계산한다 · `and` 와 `or` 는 앞에서부터이고 뒤를 건너뛸 수 있다 **언제 · 왜:** 2026-10-08 소유자 결정 — 처리기가 정한다. 같은 날 효과를 내는 피연산자를 범위에서 뺐다
FC-0002	§6.3.3	**`min` 과 `max` 에 NaN 이 끼면** **고를 수 있는 범위:** NaN 이 아닌 쪽의 값 · 또는 · NaN **정해져 있는 것:** 두 피연산자가 모두 NaN 이 아니면 결과는 하나다 **언제 · 왜:** 2026-10-08 모호 M-0003 에서 옮김 — 기계의 최소 · 최대 명령이 이 자리에서 서로 다르게 답한다
FC-0003	§6.2.6	**내용이 같은 읽기 전용 나열이 같은 자리인가** **고를 수 있는 범위:** 한 벌로 합쳐 같은 자리에 둔다 · 또는 · 따로 둔다 **정해져 있는 것:** 나열의 내용과 길이는 같다 · 둘 다 프로그램 끝까지 산다 · 두 쪽이 모두 리터럴에서 왔다는 것이 번역 시점에 보이는 `same_slice` 는 거부된다(`E-LIT-IDENTITY`) **언제 · 왜:** 2026-10-10 소유자 «이것도 열린 선택 free choice 에 들어가야지요» — 본문이 «정하지 않는다» 고만 적고 목록에 없었다. 같은 상수를 합치는 것은 링커와 C 컴파일러가 기계마다 다르게 한다
FC-0004	§6.2.4	**셈이 낸 NaN 의 부호** **고를 수 있는 범위:** 결과 NaN 의 부호 비트와 나머지 비트 **정해져 있는 것:** 결과가 NaN 이라는 것 · NaN 이 아닌 결과의 값 **언제 · 왜:** 2026-11-02 소유자 «추천대로»(W-20261102-1931) — 벡터 낱말을 세 처리기로 견준 시험에서 NaN 둘이 만나는 자리와 `0.0 / 0.0` 의 부호만 갈렸다. IEEE 754 도 정하지 않는다