Lowent

The Lowent Programming Language — Standard Specification

English edition

Document edition v0.0.32

Language edition v1.33 · built 2026-11-13

> This document defines the Lowent language itself. It should be possible to learn and implement the language from this document alone, and no place is left that can be understood only by reading another document.
>
> **This is a translation.** The normative text is the Korean edition; where the two differ, the Korean edition prevails. Clause and paragraph numbers are the same in both editions. Annexes B to F are generated from Korean tables and appear in Korean. [All chapters are current with the Korean edition.]

Contents

1 Scope 7
1.1 What this document specifies 7
1.2 What this document does not specify 7
1.3 The purpose of this language 7
1.4 How to read this document 8
1.5 Grammar notation and the frame of a clause 9
2 Normative references 11
2.1 External standards this document relies on 11
2.2 What this document does not rely on 11
3 Terms and definitions 12
3.1 The parts of a program 12
3.2 Values and types 12
3.3 Contracts and verification 14
3.4 Effects and capabilities 15
3.5 Memory and ownership 15
3.6 Processors and conformance 15
3.7 Source and notation 16
3.8 Computation and statements 17
3.9 Environments and conformance 17
4 Conformance 20
4.1 Conforming programs 20
4.2 Conforming processors 20
4.3 Implementation-defined behaviour and free choice 21
4.4 The safe subset and trust boundaries 22
4.5 Environmental limits 22
4.6 The edition of this document and the edition of the language 23
4.7 What is in this document but a processor cannot yet do 23
5 Environment 24
5.1 The translation environment and the execution environment 24
5.2 Representation of source files 24
5.3 Minimum requirements of the execution environment 25
5.4 Diagnostics 26
5.5 Program start and end 26
5.6 Machine names 27
5.7 Where a unit states what it is — `package` 31
5.8 Building differently per build — `build option` and `config` 31
5.9 Decided but not built: tools — verified versions · the package file · private sources · splitting a large build 32
6 Language 34
6.1 Lexical elements 34
6.1.1 White space and comments 34
6.1.2 Keywords 35
6.1.3 Identifiers 37
6.1.4 Literals 38
6.1.5 Dots and forms 46
6.1.6 Newline is not a closer 47
6.1.7 What is not closed 50

6.1.8 What a source shall have ... 51
6.2 Types ... 51
6.2.1 Kinds of type ... 52
6.2.2 Integer types ... 52
6.2.3 The truth-value type ... 56
6.2.4 Floating-point types ... 56
6.2.5 Conversions between types ... 56
6.2.6 Rows — arrays and slices ... 58
6.2.7 Aggregates — struct and enum ... 70
6.2.8 Types that hold an answer — result and option ... 74
6.2.9 Named types ... 77
6.2.10 Reading inside a value — field ... 79
6.2.11 Values with lanes — `vec` ... 79
6.2.12 Sets made of bits — `bitset` ... 80
6.2.13 An eye laid over bytes — views ... 81
6.2.14 Parameters with a narrowed range of values — `range` ... 83
6.2.15 Nailing down the byte layout — `layout` and `align` ... 84
6.2.16 Explicit conversion — `cast` ... 85
6.2.17 What holds characters — a string is not a type ... 90
6.2.18 Type names without meaning ... 90
6.2.19 Variants that carry values — building and taking apart ... 91
6.2.20 Seeing scattered pieces as one — `segments` ... 92
6.2.21 Fetching ahead — `prefetch` ... 93
6.2.22 What it means for types to match ... 93
6.2.23 The shape of writing fields ... 94
6.2.24 Integers of arbitrary width — `bits` ... 94
6.2.25 Decided but not built: type words — `c8`…`c64` · `b8`…`b128` · `list` · an overlapping sum · a child region ... 94
6.3 Expressions and operations ... 95
6.3.1 Prefix notation is the default ... 95
6.3.2 Infix notation — the `expr` island ... 96
6.3.3 Core ops and builtin ops ... 98
6.3.4 The rules of arithmetic ... 100
6.3.5 Bitwise operations ... 102
6.3.6 Choosing **by name** what to do on overflow ... 105
6.3.7 Ops that handle lanes ... 106
6.3.8 Asking the size — `size_of` ... 108
6.3.9 Transcendental functions ... 108
6.3.10 A truth value is not a number ... 109
6.3.11 Widths and lanes are part of the type ... 109
6.3.12 The rules inside an `expr` island ... 109
6.4 Declarations and contracts ... 110
6.4.1 Declaring an op ... 110
6.4.2 What a contract is ... 113
6.4.3 The six clauses of a contract ... 113
6.4.4 How to write contracts ... 115
6.4.5 Giving a contract a name ... 116
6.4.6 Contracts remove checks ... 116
6.4.7 The build mode decides the remaining checks ... 117

6.4.8 Names and scopes 118
6.4.9 Grades of contracts 119
6.4.10 Violations that can be decided now 121
6.4.11 What cannot be a name 122
6.4.12 What contracts refer to 123
6.4.13 Test blocks 123
6.5 Statements and control flow 126
6.5.1 Statements that make names 126
6.5.2 Conditions — `if` 130
6.5.3 Loops — `while` 131
6.5.4 A condition that leaves — `guard` 136
6.5.5 Going back — `return` 138
6.5.6 Blocks and indentation 139
6.5.7 Three ways of saying failure 140
6.5.8 Handling failure — `try` and the consuming forms 141
6.5.9 Stopping — `panic` 142
6.5.10 Loops that run divided — `parallel` 142
6.5.11 What a body shall have 143
6.5.12 Only the errors written occur 143
6.6 Telling variants apart — `match` 145
6.7 Tests — `test` and `expect` 148
6.8 Translation-time parameters — `comptime` 148
6.8.1 It shall be computable at translation time 149
6.9 Places that cannot be checked — `unsafe` and `extern` 155
6.9.1 Writing machine instructions directly — `asm` 155
6.9.2 Where C is met — the boundary of `extern` 157
6.9.3 Places that use instructions that differ by machine — `unsafe target` 160
6.9.4 Decided but not built — `raw` · `addr` · `adopt` · `clone` 160
6.9.5 Decided but not built: special cases — intrusive lists · graphs · a single-producer single-consumer queue · `layout c` 161
6.10 Modules 162
6.10.1 What is exported 162
6.10.2 What is imported 163
6.10.3 When names collide 164
6.10.4 The order of declarations 164
6.10.5 Ops inside an op — local ops 165
6.11 Type-associated ops and traits 166
6.11.1 Attaching an op to a type 166
6.11.2 Traits 166
6.11.3 Calling an attached op — method 170
6.11.4 Where attached ops go wrong 171
6.12 Flowing in one stream — `pipe` 171
6.12.1 Fusion is meaning, not optimisation 176
6.12.2 Reading only as much as needed 176
6.12.3 `map` · `filter` · `fold` outside `pipe` — ops of a library 177
7 Effects and capabilities 179
7.1 Effects 179
7.1.1 How to write the effects line 181
7.2 Capabilities 182

7.3 What purity means ... 185
7.4 Where other languages are met ... 186
7.5 Doing things atomically ... 186
7.6 Where devices are talked to ... 187
7.6.1 Register blocks — `mmio` ... 188
7.7 Where interrupts are received ... 189
7.2.1 Where capabilities mismatch ... 189
7.2.2 Where things come in and where they go out ... 191
8 Memory and ownership ... 193
8.1 Where values live ... 193
8.2 Places that receive regions ... 194
8.3 Moving and copying ... 196
8.3.1 Where a value is put — temporaries and destinations ... 196
8.4 References ... 197
8.4.1 A reference cannot outlive what it points at ... 198
8.5 Ownership and destroying ... 198
8.6 How far memory safety is guaranteed ... 202
8.7 Reading bytes again — `bit_cast` and `view` ... 202
8.7.1 `bit_cast` — the bits as they are, only the way of reading ... 202
8.7.2 `view` — laying a layout on ... 203
8.8 Changeability flows **only as written** ... 203
8.9 Same name or not, types differ ... 204
8.10 Storage the linker gives — `storage reserved` ... 204
8.11 Parentheses do not cross blocks ... 205
8.12 Exclusivity — many readers **or** one writer ... 205
8.13 Allocators — capability · policy · state ... 209
8.14 Choosing an allocator per object — the `use` clause of an op head and the `use` · `keep` of bindings ... 211
9 Library ... 214
9.1 Libraries are written in the same language ... 214
9.2 What libraries shall keep ... 214
9.3 What is language and what is library ... 215
9.4 How ripe is it — the ladder of maturity ... 216
9.5 What enters the standard library ... 217
10 Concurrency ... 218
10.1 Concurrency is an effect too ... 218
10.2 Actors ... 218
10.3 Making flows of execution ... 221
10.4 The place that binds flows — `task_group` ... 223
10.5 Delivering messages later ... 223
10.6 Exchanging between flows — channels ... 225
10.7 How far it can be used — `build profile` ... 227
10.8 What a machine can bear — `build tier` ... 227
10.9 Where running side by side goes wrong ... 228
A Annex A — Grammar summary ... 229
A.1 Keywords ... 229
A.2 The shapes of statements ... 229
A.3 Statements ... 230
A.4 Variants · tests · actors ... 230

A.5 Expressions 231
A.6 The lexical grammar of literals 231
A.7 Head words and closers 233
A.8 Removed words 234
A.9 Type words 236
A.10 The full grammar 237
A.10.1 The lexical grammar 238
A.10.2 Translation units and declarations 239
A.10.3 Statements 240
A.10.4 Expressions 241
A.10.5 Types 242
B 부록 B — 진단 목록 (Annex B: Diagnostics) 243
B.1 실행 중 트랩 (E-VM-...) 243
B.2 경고 (W-...) 243
B.3 번역 잘못 (E-...) 243
C 부록 C — 용어 색인 (Annex C: Index of terms) 247
D 부록 D — 기본 연산과 빌트인 연산 (Annex D: Core ops and builtin ops) 251
D.1 이름과 뜻 251
D.1.1 기본 연산 (core op) 251
D.1.2 빌트인 연산 (builtin op) 255
D.2 받는 것과 내는 것 259
E 부록 E — 모호 목록 (Annex E: Open issues) 273
F 부록 F — 자유 선택 목록 (Annex F: Free choices) 275

1 Scope

1.1 What this document specifies

1 This document defines the *Lowent* programming language. It specifies the following four things.

2 a) The *representation* of programs written in Lowent — which characters, arranged how, form a program.

3 b) The *syntax* and *constraints* of Lowent programs — which arrangements are correct and which are not.

4 c) The *semantic rules* for interpreting Lowent programs — what a correct program does.

5 d) The *conformance* requirements that a *processor* which translates and executes Lowent programs shall meet.

> **In plain words**
>
> This document is the only document that answers *"what is Lowent"*. Someone learning the language and someone building a new Lowent processor both need to read only this one document. No place has been left where another document must be found and read.

1.2 What this document does not specify

1 The following are outside the scope of this document. Outside the scope means *"not specified here"*, not *"forbidden"*.

2 a) How Lowent programs are made — how to use editors, build tools and package managers.

3 b) The internal structure of a particular processor — which intermediate representation it uses, how it optimises.

4 c) The complete list of the standard library — §9 specifies the *rules* a library shall follow, not the list of individual functions.

5 d) The behaviour of operating systems and hardware — this document states only what it requires of them (§5).

> **In plain words**
>
> *"Not specified"* and *"forbidden"* are easy to confuse. For example, this document does not specify an editor, but that does not mean you must not use one; it means **it is not this document's business**.

1.3 The purpose of this language

1 Lowent is a systems programming language designed to lower *semantic entropy*.

> **semantic entropy** (의미 엔트로피)
>
> The **uncertainty that remains** about a program's actual behaviour after its whole source has been read. Put another way, the **amount of information that must be brought in from outside the source** to know what the code does.

2 Every rule of this language was chosen to reduce that uncertainty. Specifically, it removes the following four things.

3 a) **Hidden behaviour** — nothing happens that the source does not state. There is no invisible memory allocation, no invisible control flow and no invisible type conversion.

4 b) **Hidden cost** — how expensive a line is, is written on that line.

5 c) **Hidden assumptions** — what a function requires is written as a *contract*, and the processor checks it (§6.4).

6 d) **Overlapping notation** — the same meaning cannot be written in several ways. Different notation means different meaning.

7 Low semantic entropy **does not mean that the source corresponds one-to-one to machine code.** There are places where, for efficient implementation and optimisation, the processor must have a choice.

8 This language **marks such places and isolates them from the rest.** Every place where the processor chooses is marked (**free choice**, §4.3), and each place states the range it may choose from. The processor makes no choice at an unmarked place.

9 So the person writing a program **can know what is fixed and what is open.** At an open place they **may choose for themselves** whether to leave it to the processor or to rewrite it in a fixed form.

> **In plain words**
>
> Entropy is the amount of «what stays unknown». The mere existence of choices does not increase entropy — what increases it is **not knowing where** the choices are, and **not knowing how far** what can happen extends.

> **In plain words**
>
> A language close to the machine cannot tie the processor down so that it chooses nothing. Tied that way, it could not use a faster path even when it knew one. Conversely, if places of choice are scattered all over the language, every line read raises the question «does the processor choose something here too». That question is the entropy.

> **In plain words**
>
> So this language **gathers the places of choice in one list** (annex F), **puts a mark** at each such place in the text, and **states the range** of each. On a line without a mark the reader can trust that what is written happens as written; on a line with a mark they know exactly what is open. Even if some uncertainty remains, **knowing its place and its size** — that is what this language aims at.

> **Note (informative)**
>
> The name «Lowent» is short for **low entropy**. The name is the criterion — whether a feature goes in is decided by *"does it increase semantic entropy"*.

1.4 How to read this document

1 This document distinguishes the *normative* from the *informative*.

> **normative** (규범)
>
> Text that defines the language. Processors and programs shall obey it. Unless marked otherwise, the whole body of this document is normative.

> **informative** (참고)
>
> Text that helps understanding. It does not define the language. Boxes marked «Note», «In plain words» and «Caution» are informative. Where the normative and the informative appear to disagree, **the normative is right**.

2 Clauses are numbered. A number is an **address**, and the meaning of a number once given does not change. When another document cites this one, it cites by number (for example §6.4.2).

2a A number is a sequence of integers joined by dots (annexes begin with a capital letter). **No other form is used** — in particular, numbers such as `3a` are not made in order to insert something in between. If a clause must be inserted, **the following numbers shall be shifted**, and that can be done only before this document is widely cited.

> **Note (informative)**
>
> There really is *"a last moment at which numbers can be shifted"*. While this document is young, moving numbers is cheap, but once other documents and tools start pointing at these numbers it becomes expensive. That is why this rule is worth something only if it is kept **now**.

3 Paragraphs within each clause are numbered too (1, 2, 3 …). A precise citation gives the clause number and the paragraph number together (for example 6.4.2 paragraph 3).

4 A term is given with its English form in parentheses, and its meaning, where it first appears. The definitions of all terms are gathered in §3.

> **Caution — The meaning of the modal verbs used in this document**
>
> «shall» (Korean 「-하여야 한다」 · 「-이어야 한다」) is a **requirement**. Not meeting it means not conforming. «may» (「-할 수 있다」) is a **permission**. «should» (「-하는 것이 좋다」) is a **recommendation**; not following it still conforms. «shall not» (「-하지 아니한다」) is a **prohibition**.

1.5 Grammar notation and the frame of a clause

1 This document writes its grammar as *productions*. There is one notation, the same throughout the document.

> **production** (생성 규칙)
>
> A statement of which sequence of units a unit of the grammar consists of. Its form is `name ::= body`.

Table 1 — Grammar notation

Notation	Meaning
`name ::= body`	`name` consists of `body`
`A B`	`A` followed by `B`
`A ¦ B`	`A` or `B`
`[ A ]`	`A` absent or once
`{ A }`	`A` zero or more times
`( A )`	grouping
`"text"`	those characters, literally
`"a" … "z"`	one of the characters from `a` to `z`
`? text ?`	a meaning stated in words. The words are normative
`name`	refers to another production

2 One or more is written `A { A }`. No other notation is used.

3 Production names are written in lowercase English letters and hyphens (`if-statement`). The same name refers to the same rule throughout the document.

3a The grammar has **two layers**. The *lexical grammar* specifies how a sequence of characters is divided into tokens (§6.1). The *syntactic grammar* specifies how a sequence of tokens is read as grammatical elements.

> **lexical grammar** (어휘 문법)
>
> The rules that divide a sequence of characters into a sequence of tokens. The terminal symbols of its productions are **characters**.

> **syntactic grammar** (구문 문법)
>
> The rules that read a sequence of tokens as grammatical elements. The terminal symbols of its productions are **tokens**.

3b `"if"` in the syntactic grammar means one token with that spelling. White space and comments between tokens are discarded at the lexical stage and do not appear in the syntactic grammar.

4 What a production specifies is **form** only — which sequence of words is read as that grammatical element. Which of the well-formed ones conform is not specified by productions but by constraints and static semantics.

5 A condition that cannot be written as a production is written as a numbered paragraph directly below that production.

6 A clause that specifies a construct of the language has the following sections **in this order**. A section with nothing to say is omitted.

Table 2 — The frame of a clause

Section	What it states
Syntax	Productions. They specify form only
Constraints	Conditions a well-formed program shall meet to conform. A program that violates one does not conform
Static semantics	What is determined at translation for a conforming program — the type of an expression, what a name refers to
Dynamic semantics	What happens at execution — the order of evaluation, changes of state, stopping
Diagnostics	The diagnostic identifier a processor issues when a constraint is violated
Examples	Conforming examples and rejected examples. Each states which paragraph it tests
Notes	In plain words · Caution · the reasons for a choice. Not normative (⟦§1.4⟧)

7 A constraint states only the requirement. The diagnostic identifier for violating it is given in the diagnostics section — the diagnostic identifier does not define the requirement.

8 Paragraph numbers do not restart in each section; they continue through the clause.

9 Some clauses have not yet been moved into this frame. Such clauses keep their earlier wording, and their normative force is the same.

> **Note (informative)**
>
> A «grammar shape» box is a form to see at a glance. The definition of the grammar is the productions in the syntax section; where the two appear to disagree, the production is right.

> **In plain words**
>
> The sections are separated because **whom they speak to** differs. Syntax speaks to the one who reads text as a tree (the parser), constraints and static semantics to the one who checks whether that tree is correct (the checker), and dynamic semantics to the one who runs the program. If the three are mixed in one paragraph, the reader has to separate them, and every reader separates them differently.

2 Normative references

2.1 External standards this document relies on

1 The following documents form part of this document to the extent that this document cites them. For those with an edition given, that edition applies; for those without, the latest edition applies.

Table 3 — Normative references

Document	What it is relied on for
ISO/IEC 10646 (Unicode)	The character set of source files. A Lowent source is a Unicode string encoded in UTF-8 (§5.2).
IEEE 754-2019	The representation and operations of floating-point numbers. `f32` · `f64` follow it (§6.2.4).
ISO/IEC 9899 (C)	The boundary exchanged with the C language (§7.4). Lowent does not define the meaning of C; it **cites** the C side's promises.

> **Note (informative)**
>
> Having C as a normative reference does not mean that Lowent is built on top of C. It is because that standard already fixes the promises a Lowent program shall keep **when it meets code written in C inside the same machine**.

2.2 What this document does not rely on

1 Lowent does not rely on the following. This is a portability requirement of the language.

2 a) A particular operating system — the whole language part of this document holds even in an environment without an operating system (*freestanding*) (§5.1).

3 b) A particular CPU architecture — but the **minimum requirements** specified by §5.3 are required.

4 c) A run-time support library — the language itself requires no runtime.

5 d) A *garbage collector* — this language has none (§8).

> **In plain words**
>
> What this list means is this — **a computation written in Lowent runs as it is on a small device without an operating system.** Only outside work such as reading a file needs the environment's permission, and that permission is handed over as a capability (§7.2).

> **Note (informative)**
>
> It is no accident that there are only three normative references. The more there is to rely on, the fewer the places this language can run.

3 Terms and definitions

1 In this document the following terms are used only in the meanings given here. They may differ from their everyday meanings or from their meanings in other languages; whenever a meaning is in doubt, this clause is the reference.

> **In plain words**
>
> A first-time reader need not read this clause from start to end. When an unknown word turns up in the text, come back here. Each term carries the number of the clause that actually uses it.

3.1 The parts of a program

> **program** (프로그램)
>
> The set of Lowent source files that are translated together and form one unit of execution. The exact composition is specified by §5.1.

> **module** (모듈)
>
> One named source file. A module writes its own name on its first line and marks what it exports to other modules. A module name need not be the same as the file name.

> **op** (op)
>
> A unit of execution with a name and a contract. It corresponds to a «function» in other languages, but in Lowent **its kind depends on whether it has side effects** (§6.4.1): one that produces no effect at all is `fn`, one that produces effects is `proc`.

> **declaration** (선언)
>
> A construct that gives a name a meaning. It says what exists.

> **definition** (정의)
>
> A declaration that also supplies the entity itself. The definition of an op includes its body.

3.2 Values and types

> **value** (값)
>
> What a computation produces. Every value has exactly one type.

> **type** (타입)
>
> A set of values together with the operations allowed on them. Every Lowent type **has a fixed size and representation** — there is no place the processor decides at will (§6.2).

> **width** (폭)
>
> The number of bits an integer type occupies. The width of `u8` is 8. The width of a type with a number in its name is **the same on every processor**. Only `usize` · `idiff` are exceptions; their width is chosen by the implementation — at least 16 bits, usually the address width of the target machine (§6.2.2).

slice (슬라이스)

A reference to a range of values of the same type laid out contiguously. It carries a start position and a **length** together. Carrying the length is what makes it differ from a pointer, and is why its bounds can be checked (§6.2.6).

character literal (문자 리터럴)

One character in single quotes (`'a'`). Its value is that character's code, an integer. What does not fit in one slot is not a character literal (§6.1.4).

string literal (문자열 리터럴)

A row of bytes in double quotes (`"abc"`). `len` gives **the number of elements** — the prefix decides what an element is.

escape (이스케이프)

A notation inside a character or string, starting with a backslash, that stands for one character (`\n`). There are **only fourteen** (§6.1.4).

prefix (접두사)

A word placed before a literal that decides what its elements mean. There are **only two**: `u` (UTF-16 code unit) · `U` (code point). Without a prefix the elements are bytes.

text literal (텍스트 리터럴)

A string that holds several lines as they are. A tag is written after `text`, and the literal ends at a line on which the same tag stands alone from the very first column. Escapes in the body are not processed. It corresponds to what other languages call a heredoc, nowdoc or delimited string.

floating-point literal (부동소수점 리터럴)

A number with a decimal point or an exponent written directly in the source (`1.5` · `1e3` · `0x1p3`). There shall be digits on both sides of the point — `.5` and `1.` are not floating-point literals (§6.1.4).

literal (리터럴)

A value written directly in the source. `42`, `true`, `"text"` and the like.

enum (열거)

A value that is **exactly one** of a set of named variants. Each variant may carry a value. Since the variant name says what the value is, some bit patterns remain unused — so an enum has «trap representations».

trap representation (덫 표현)

A bit pattern that is **not** a value of its type. Reading such a bit pattern as a value of that type has no meaning, and this language does not allow such a read (§8.7.1). A scalar type with no trap representation at all is called `plain`.

brand (상표)

A **name that separates** storage. So that values with the same representation but from different storage do not mix, a type made with `newtype` is used as the name of that storage. One brand designates one storage (§6.2.9).

3.3 Contracts and verification

contract (계약)

The **promise an op writes for itself** about its input and output. It consists of six clauses — `requires` (what shall be true on entry) · `ensures` (what is true on exit) · `errors` (when it fails) · `effects` (what effects it produces) · `access` (what it touches, and how) · `tests` (examples). The full definition is in §6.4.3.

precondition (사전 조건)

What the `requires` clause of a contract says. A condition that shall be true at the moment of entering the op.

postcondition (사후 조건)

What the `ensures` clause of a contract says. A condition that is true when the op ends normally.

check (검사)

Confirming during execution that a condition is true. If the processor **proves that the condition is always true**, it removes the check. If it cannot remove it, the check remains (§6.4.6).

trap (트랩)

Stopping the program at once when a condition is broken. Lowent does not continue computing in a wrong state — it stops rather than return a wrong answer.

undefined behaviour (미정의 동작)

Behaviour the source does not determine. There is none in the **safe subset** (§4.4) — within it, every situation this document does not define ends in a diagnostic (§5.4) or a trap. This is where it differs most from C.

safe subset (안전한 부분집합)

The set of programs that do not use `unsafe` and `extern`. Within this set there is no undefined behaviour.

build mode (빌드 모드)

The setting that decides whether contract checks that could not be proved stay during execution or are removed. It is written in the source as `build <mode> .` (§6.4.7).

trust boundary (신뢰 경계)

The place where the language's guarantee ends and a person's promise begins. `unsafe` and `extern` mark that place in the source (§6.9).

3.4 Effects and capabilities

effect (효과)

An influence an op has on the outside world. Reading a file, looking at a clock, allocating memory, creating another flow of execution and the like are each an effect. An op shall write the effects it produces in its contract, and if it produces more effects than written, translation is refused (§7.1).

pure (순수)

Producing no effect at all. A pure op always gives the same result for the same input.

capability (권한)

A **value** that stands for the right to produce an effect. Lowent has no global capability anyone may use — to read a file, a file capability shall be **handed over** as an argument (§7.2).

3.5 Memory and ownership

region (영역)

A section of memory where values live. For each value, the region it lives in is fixed together with its type (§8.1).

reference (참조)

Something that points at a value elsewhere. Read-only references and write references are distinguished, and for one value there can be only one write reference at a time (§8.4).

ownership (소유)

Who is responsible for destroying a value. Lowent tracks owners statically, so destroying twice or forgetting to destroy is refused at translation (§8.5).

escape (탈출)

A reference to a local value outliving that value. Lowent refuses this at translation (§8.4.1).

3.6 Processors and conformance

processor (처리기)

The general name for whatever translates or executes Lowent programs. A compiler, an interpreter, and something that does both are all processors. This document does not specify their internals; it specifies only **observable behaviour**.

translation (번역)

Turning a source into a form that can be executed. Errors found during translation are reported as diagnostics.

diagnostic (진단)

A message by which a processor reports an error. It has a stable identifier (for example `E-TYPE-INSTANCE`), and the meaning of that identifier does not change across editions (§5.4).

conforming program (적합한 프로그램)

A program that keeps all the syntax and constraints of this document (§4.1).

conforming processor (적합한 처리기)

A processor that translates and executes conforming programs as this document specifies, and issues diagnostics for programs that do not conform (§4.2).

3.7 Source and notation

Lowent (로우엔트)

The programming language this document defines. The name is short for **low entropy**.

token (토큰)

The smallest piece a source is divided into. Keywords, identifiers, literals and punctuation are tokens.

keyword (낱말)

A name whose meaning the language has fixed. It cannot be used with another meaning. The list is in annex A.

identifier (이름)

A name the author makes. It consists of ASCII letters, digits and underscores — **it contains no dot.** A dot is the mark that joins two names when choosing a field of a value or naming a module (§6.1.3).

comment (주석)

From `rem` to the end of the line. The processor treats it as absent.

form (폼)

One unit of prefix notation. A closer marks its end.

closer (닫개)

The mark that says a form has ended. It is **one** full stop `.`. A parenthesis `)` also closes the forms opened inside it, and `end` closes only its own `do` (§6.1.6).

newline (개행)

The place where a line changes. It is treated exactly like a space and does **not** close a form (§6.1.6).

prefix notation (전위 표기)

A notation in which the name of an operation comes first and its arguments follow. There are no precedence rules.

infix notation (중위 표기)

A notation in which the operator symbol comes between its arguments. It can be used only inside an `expr` island.

island (섬)

The place where infix notation is switched on. It begins with `expr`.

byte order mark (BOM)

A mark some editors put at the start of a file (U+FEFF). It cannot appear in a Lowent source.

3.8 Computation and statements

expression (식)

Something that produces a value.

statement (문장)

Something that is executed. An expression makes a value; a statement does work.

core op (기본 연산)

An operation whose meaning the language fixes and which is called anywhere by its name alone. It shares one namespace with the author's names, so it cannot be hidden by them.

builtin op (빌트인 연산)

An operation whose meaning the language fixes and which is called only after `call_builtin`. It has its own namespace, so it does not collide with the author's names. It is for building system libraries and processors; ordinary programs use the libraries that wrap it.

short-circuit (단락 평가)

Not computing the second part when the first part alone decides the result. `and` and `or` do this.

widening (넓히기)

A type conversion that loses no value. It may happen automatically.

narrowing (좁히기)

A type conversion that may lose a value. It shall be written in the source, and traps if the value does not fit.

trait (트레이트)

A list of ops a type shall provide. If a type is written as providing it, the processor checks. It makes no hierarchy among types — this language has no inheritance.

actor (액터)

A unit of execution with its own state. The state exists only inside the actor.

3.9 Environments and conformance

translation environment (번역 환경)

The place where a source is turned into a form that can be executed.

execution environment (실행 환경)

The place where the result actually runs.

entry point (시작점)

The op at which an executed program starts. Its name is `main`.

exit status (종료 상태)

The value the entry point returns. 0 means success.

library module (라이브러리 모듈)

A translation unit without an entry point. Other programs take it and use it.

conformance (적합성)

Whether what this document specifies has been kept. Used of programs and of processors.

syntax (구문)

The rules about which arrangements of characters are programs.

constraints (제약)

Rules that shall be kept even when the syntax is right. Violating one produces a diagnostic.

semantic rules (의미 규칙)

The rules about what a correct program does.

representation (표현)

Which bits a value is laid out as inside the machine.

octet (옥텟)

8 bits. In this language a byte is always an octet.

two's complement (투스 컴플리먼트)

A way of representing signed integers. It is the only way this language requires.

place (자리)

Somewhere a value **can be written**. A local variable, a field of a struct, one slot of a row are places. What can stand on the left of `set` is exactly a place — something that can only be read is not a place.

qualification (한정)

Fixing which name is meant by putting in front of a name the place where it lives. In this language qualification is written with a dot **attached** to the name, and it has only three meanings: a module, a variant, and a declaration belonging to a type. Reading the inside of a value is not qualification.

destination (목적지)

The place where a value is made and put. If the three conditions of §8.3.1 hold, the value is made here **directly**, without an intermediate temporary.

temporary (임시값)

A value in a nameless place, made when there is no destination or the conditions do not hold. It lives in the smallest enclosing scope, and when the scope ends the temporaries are destroyed **in the reverse order of their making**.

garbage collector (가비지 컬렉터)

A mechanism that finds unused memory during execution and gives it back. **This language has none.**

stage (스테이지)

A place inside a `pipe` of §6.12 that takes one element and passes it on. The vocabulary is closed (seven), so no other operation can be a stage.

terminal (종결자)

The place that ends a `pipe` and gives its result. A `pipe` has **exactly one** terminal.

context (문맥)

A value attached with `with` to a stage or terminal of a `pipe`, computed once before the traversal and handed to the op as its last argument (§6.12 (7)). Only a value without references can be a context.

local op (로컬 op)

An op declared directly below an op body and visible only inside that op (§6.10.5). It does not see outer names, so it is not a closure.

optimisation (최적화)

The processor's discretion to make a program faster or smaller without changing its meaning. **Being discretion, a program shall not rely on it** — a property that must be relied on is written as meaning, not as optimisation (the fusion of §6.12 is an example).

4 Conformance

4.1 Conforming programs

1 A *conforming program* is a program that keeps all the syntax (§6) and constraints (§6) of this document.

2 A conforming program does not rely on behaviour this document does not specify.

2a Diagnostics come in two kinds: **those that stop translation** (their names begin with `E-`) and **those that do not** (`W-` · `NOTE-`). Only the former decide conformance; a program can conform even if the latter are present.

2b That is, a conforming program is a program with **no diagnostic that stops translation**.

> **Example — a conforming program**
>
> **Source**
>
> ```
> module ex_conform .
>
> rem This program keeps everything this document specifies.
> 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 .
> ```

> **In plain words**
>
> Called as `clamp 200`, the program above gives 200. Called as `clamp 300` — since the contract said *"at most 255"* — it **stops right there** (`E-VM-CONTRACT`). Not returning a wrong answer is this language's promise, and that promise is kept at execution.

> **Caution — Conforming and correct are different things**
>
> A conforming program can still give a wrong answer. Conformance is **whether the rules of the language were kept**, not **whether it does what was intended**. The latter is handled by contracts (§6.4.3) and tests.

4.2 Conforming processors

1 A *conforming processor* shall satisfy all of the following.

2 a) It shall translate and execute conforming programs with the meaning this document specifies.

3 b) It shall issue **at least one diagnostic** for a program that does not conform. Whether translation continues after a diagnostic is decided by the processor.

4 c) Where it uses a diagnostic identifier this document specifies, it does not use it with a changed meaning.

5 d) Where it provides extensions this document does not specify, they do not change the meaning of a conforming program that does not use them.

> **Note (informative)**
>
> b) does not mean *"it shall find every error"*. For one non-conforming program, it is enough that **at least one** diagnostic is issued. Since Lowent's principle is not to postpone checks, however, real processors usually collect and report all errors.

4.3 Implementation-defined behaviour and free choice

1 The places where this document leaves discretion to the processor are of **two kinds**, and those places are **explicitly marked**.

> **implementation-defined** (구현 정의)
>
> Something the processor decides but **shall document**. Example: the maximum recursion depth. A program can ask for the value, and it may differ between processors.

> **free choice** (자유 선택)
>
> A place where **the choice must be left free** for the sake of optimisation or of the processor's circumstances. The processor chooses one of two or more behaviours or results this document allows at that place. The processor **need not document** which it chose.

1a For each place of **free choice** this document states **the range that may be chosen from**. The processor chooses only within that range.
1b The processor may choose differently at the same place of the same program at each translation and each execution.
1c **free choice is not something left unspecified.** Whichever member of the range is chosen, it is behaviour this document specifies.
1d A place of **free choice** is marked in the text with a «free choice» box. Annex F gathers all of them, and a place not in annex F is not **free choice**. Adding a place or widening a range is a revision of this document.
1e A conforming program does not rely on **which one** the processor chooses at a **free choice** (§4.1 (2)).
1f There are places this document **has not yet decided**. Such places are not discretion — they are not a choice given to the processor but something this document must decide later. They are marked in the text with an «open issue» box and gathered in annex E.

> **open issue** (모호)
>
> A place this document **has not yet decided and must decide later**. The possible readings are written side by side. Once decided, it becomes a normative paragraph, or, if it is decided to leave it open, a free choice.

1g An open issue and a **free choice** are different. An open issue is **undecided**; a **free choice** is **a decision** — a decision to leave it open. A conforming program does not rely on the place of an open issue either (§4.1 (2)).
2 There is no discretion outside these two kinds. In the **safe subset** (§4.4) undefined behaviour does not exist — no program in it is executed in a state of «nobody knows what will happen».

> **In plain words**
>
> In plain words the two kinds are these. **Implementation-defined** is «it may differ between processors, but the processor writes down what it chose». **Free choice** is «any of these may be chosen, and the choice need not be written down».

> **In plain words**
>
> Free choice exists so that the processor can take a faster path. For example in `add f x. . g y. . .` the answer is the same whichever of `f` and `g` is computed first. If the order were nailed down, the processor could not use a faster order even when it knew one. So that order is left open.

> **In plain words**
>
> Leaving something open does not mean anything at all may happen. Each place **states the range that may be chosen from**, and nothing outside it happens. And such places are exactly the list in annex F — at a place not on the list the processor chooses nothing at will. Ranges are fixed narrowly, place by place.

> **In plain words**
>
> The one thing a person writing a program needs to remember: **do not rely on which one is chosen.** If the order matters, write it in a form whose order is fixed (the box in §6.3.1 says how).

4.4 The safe subset and trust boundaries

1 The *safe subset* is the set of programs that do not use `unsafe` and `extern`. The guarantees of the previous clause are about this set.

2 `unsafe` and `extern` mark a *trust boundary* (§6.9). At that place the language's guarantee ends and **a promise written by a person** begins.

3 If the promises beyond the boundary are true, the behaviour of the whole program is as this document specifies. If a promise is false, this document guarantees nothing — what happens then is not determined.

4 So boundaries **shall be narrow**. The wider they are, the larger the area this document's guarantees do not reach.

> **In plain words**
>
> This clause exists for the sake of honesty. One cannot call a function written in another language and say *"our language has no undefined behaviour"* — this document does not know what that function will do. What can be said is **where the language's guarantee stops and where a person's promise begins**, and `unsafe` and `extern` **leave that place in the source**. Being able to find it later is as important as the guarantee.

4.5 Environmental limits

1 A processor may have resource limits. But when it hits a limit it **shall not silently truncate** — it shall issue a diagnostic or trap.

2 A processor shall document the limits it supports (implementation-defined).

3 The diagnostics issued when a limit is hit are grouped into families — when a table inside the processor overflows, `E-IR-…` · `E-TYPE-LIMIT` · `E-PARSE-LIMIT` are issued; when a resource runs out during execution, `E-VM-…`. This document does not list them one by one; **what is required is (1) and (2)** — not truncating silently, and documenting the limits.

> **In plain words**
>
> The names of the diagnostics are not nailed into the norm one by one because they are **the processor's internal affairs**. Splitting one table into two changes the names too, and if the norm had to be revised each time, that norm would be no more than a shadow following the processor. What the norm shall fix is **what is promised**, and the promise is two things: «say so when it overflows» and «write down how far it goes».

> **Caution —**
>
> Writing down a limit and **keeping** a limit are different. Writing it down but truncating silently on overflow violates (1); keeping it but not writing it down means the user learns it existed **only after hitting it**. Both are needed.

> **Note (informative)**
>
> *"Not truncating silently"* is the point of this clause. Every processor has limits. The problem is going past a limit and **carrying on without a word**; the program's result then differs from what the source said — and that is semantic entropy (§1.3).

4.6 The edition of this document and the edition of the language

1 This document has **an edition number separate** from the language. The «document edition» on the cover is this document's edition, and the «language edition» is the edition of the language this document describes.

2 A rise in the last digit of the document edition means **the text grew or was corrected**, not that the language changed. When the language changes, the language edition rises, and the middle digit or higher of this document rises with it.

4.7 What is in this document but a processor cannot yet do

1 There may be things this document specifies that a processor **cannot do yet**. Then the processor **does not treat them as if they did not exist.**

2 At such a place the processor shall say *"this word is in the specification, and what cannot do it is the processor"*. That is, it is not an error of the program.

3 This distinction is normative because **what a person does next differs**. Told *"unknown name"*, a person goes looking for a typo that is not there; told *"not yet"*, they look for another way or wait. One diagnostic decides half a person's day.

4 A processor shall make such places **countable** — asked which ops passed every static check but cannot be lowered to execution, it shall answer.

> **In plain words**
>
> This is not a clause that «excuses incompleteness». It is the opposite — a clause that says **show** the incompleteness. Between a processor that silently cannot do something and one that says it cannot, there is a difference in trust.

5 Environment

1 This clause specifies **the place where Lowent programs are translated and executed**. The grammar of the language is specified by §6 here the ground on which that grammar stands is specified.

5.1 The translation environment and the execution environment

1 The *translation environment* is the place where a Lowent source is turned into a form that can be executed. The *execution environment* is the place where the result actually runs. The two may be the same machine or different machines.

> **translation unit** (번역 단위)
>
> One source file translated together. In Lowent a translation unit is exactly one module, and a module writes its own name on its first line.

> **program** (프로그램)
>
> The set of translation units bound together into one unit of execution. A program shall have exactly one entry point.

2 Execution environments are of two kinds.

> **hosted** (호스티드)
>
> An environment with an operating system. Files, clocks, standard input and output and the like can be received and used as capabilities.

> **freestanding** (프리스탠딩)
>
> An environment without an operating system. The language itself holds unchanged; only the capabilities the host gives (§7.2) cannot be received.

3 §6 to §8 of this document hold **in both environments**. What differs between environments is only which capabilities can be received.

> **In plain words**
>
> Put simply — a computation written in Lowent has **the same grammar and meaning** even on a small device without an operating system. What differs is only the part that touches the outside world, such as «can it read a file».

5.2 Representation of source files

1 A Lowent source file shall be a Unicode string encoded in UTF-8. Other encodings do not conform, and the processor shall issue a diagnostic.

2 The end of a line is represented by the line feed character (U+000A). The processor may accept a carriage return (U+000D) in front of it, and then treats the two characters as one line end.

3 A byte order mark (*byte order mark*, U+FEFF) cannot appear anywhere in a source file. Even at the very start it is a **lexical error**, and the processor shall issue a diagnostic.

4 Only ASCII characters can be used in an *identifier* (§6.1.3). Unicode characters can be used freely in the **contents** of comments, strings and text literals.

5 The encoding in which the source is read (the bytes the processor reads) and the encoding of the values literals have (the bytes the program handles) are **separate**. The source is always UTF-8, and a literal may have as its value bytes that are not UTF-8.

> **Caution — Identifiers being ASCII is a decision, not a restriction**
>
> Allowing Unicode identifiers creates **names that look the same but are different** (the Greek letter A and the Latin A, two Hangul spellings in different normal forms, and so on). A person reading the source then reaches a wrong conclusion, and that is the uncertainty this language sets out to remove (§1.3). So identifiers are fixed to ASCII, and text meant for people is written in Unicode in comments and strings.

> **In plain words**
>
> A function cannot be named in Hangul, but comments and strings can be written in Hangul as much as you like. The examples in this repository are in fact written that way.

5.3 Minimum requirements of the execution environment

1 A conforming execution environment shall satisfy the following.

2 a) A byte (*octet*) shall be exactly 8 bits.

3 b) Integers shall be represented in two's complement (*two's complement*).

4 c) The address width shall be 16 bits or more.

4a **A pointer is not the same thing as an address.** A pointer value may carry bounds or permissions in addition to an address, so its size **may be larger** than the address width. The size and alignment of a pointer value, and whether it is a *capability* (a value that cannot be made from an integer and is derived only from another valid pointer), are determined by the execution environment (implementation-defined). A program asks for these values at translation time with `machine.addr_bits` · `machine.ptr_bytes` · `machine.ptr_align` · `machine.ptr_caps` (`machine.ptr_width` is the same as `machine.addr_bits`).

> **capability** (능력)
>
> A pointer value that carries, together with an address, **the range it may reach and what it may do** (read · write · execute). The hardware separately remembers whether the value is valid, so it cannot be made or rewritten from an integer and is obtained only by **narrowing** another valid capability. Reaching outside the range stops execution.

4b Values of this language **do not expose pointers as integers.** What points is a slice · a reference · a capability, and no integer type can hold a pointer and bring it back to life. So even on a machine whose pointers are capabilities the meaning of a program does not change.

> **Note (informative)**
>
> Machines whose pointers are larger than their addresses really exist — capability-pointer machines (64-bit addresses · 128-bit pointers, or 32-bit addresses · 64-bit pointers) carry bounds and permissions in the pointer and make pointers that went through an integer unusable. There are also machines whose pointer size differs per address space (microcontrollers with separate program and data memory, the spaces of accelerators). (4a) · (4b) are written so that, on the day such a machine is met, the language's promises do not break first.

4c **The multiplier width** is the widest width (8 · 16 · 32 · 64 bits) at which the product of two integers, up to its upper half, is produced by one instruction. It is determined by the execution environment (implementation-defined), and a program asks for it at translation time with `machine.mul_bits`. A translator may provide a means of changing that value for one translation. It may differ from the address width. The processor also fixes the **limb width** used for big-number arithmetic and answers it with `machine.limb_bits` (16 · 32 · 64 bits, implementation-defined) — the value measured and fixed on the machine if there is one, otherwise the multiplier width (16 bits if

the multiplier width is narrower than 16 bits). A translator may provide a means of changing the limb width for one translation. The multiplier width is a fact of the machine and does not change.

> **Note (informative)**
>
> Big-number arithmetic splits numbers into limbs and multiplies limb by limb. Which limb width is fast is decided not by the address width but by the multiplier — some machines with 32-bit addresses cannot produce the upper half of a 32×32-bit product (Cortex-M0). But the measured result can differ from the multiplier width (on a 32-bit x86 build, 64-bit limbs were faster), so the chosen width is kept separately. The standard library's big-number module `bigint` chooses its limb width with `machine.limb_bits` and gives the same answer at every width (RFC-0152).

5 d) For a program that uses floating-point numbers, the execution environment shall provide the IEEE 754 binary 32-bit or 64-bit format. A program that does not use floating-point numbers is unaffected by this requirement.

> **Note (informative)**
>
> Almost every machine in use today already satisfies a)–c). This document **writes them down** because otherwise «where does this language run» would become knowledge outside the source. Some signal-processing devices whose byte is not 8 bits are not targets of this language.

5.4 Diagnostics

1 The processor shall issue a *diagnostic* for a program that does not conform.

2 A diagnostic shall have a **stable identifier**. Identifiers consist of capital letters and hyphens (for example `E-TYPE-INSTANCE`), and the meaning of an identifier, once fixed, does not change.

3 A diagnostic shall give the **place** (file · line) where the error occurred.

4 Diagnostic identifiers fall into the following kinds.

Table 4 — Kinds of diagnostic identifier

Prefix	Meaning	Examples
`E-`	Error. Translation does not succeed	`E-TYPE-INSTANCE`, `E-EFFECT`
`W-`	Warning. Translation succeeds, but the author should look	`W-EXPORT-NOSYM`
`E-VM-`	Trap during execution	`E-VM-CONTRACT`, `E-VM-BOUNDS`

5 A trap during execution stops the program at once and reports the fact to the execution environment. Computation does not continue after a trap.

> **Caution — The text of a diagnostic message is not a contract**
>
> What is a contract is the **identifier**; the sentence people read is a convenience. Someone building tools should look at the identifier, not pick the sentence apart with regular expressions — the sentence may be polished.

5.5 Program start and end

1 **A program that is executed** shall have exactly one *entry point*. The entry point is the op named `main`.

1a A translation unit without an entry point also conforms — it is a *library module* other programs take and use. An entry point is needed when an executable result is made.

2 The value the entry point returns is the program's *exit status*. 0 means success, and the meaning of other values is determined by the execution environment (implementation-defined).

2a What can be used as the output of the entry point: a small unsigned integer (`u8` · `u32` and the like) · `void` (success when it ends) · `result` (0 on success; on error, the status that error determines).

2b If the value is wider than the execution environment's exit status, how it is narrowed is implementation-defined.

3 In a hosted environment the entry point may receive as arguments the capabilities the execution environment gives. Which capabilities it receives is written by the entry point in its own contract, and the execution environment hands over only what is written (§7.2).

4 When the entry point returns, the program ends. Resources open before the end shall already have been closed according to the ownership rules (§8.5); if a resource is left unclosed, the program is refused at translation.

> **Example — the entry point receives capabilities**

Source

```
module ex_env .

use stdio .

rem A capability not received exists nowhere in this program.
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 .
```

> **Rejected example — the entry point returns `u8`**

Source

```
module ex_entry .

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

Diagnostic: `E-ENTRY-OUTPUT`

> **In plain words**

The last paragraph is where this language differs most from others. It is often said that «when a program ends, the operating system cleans up», but Lowent **refuses at translation the very fact of having forgotten to clean up**. So the same code holds even in an environment without an operating system.

5.6 Machine names

1 The processor knows which *target* it builds for. Machines differ in word size, byte order, the presence of floating-point numbers and so on, and where those differences reach the meaning of a program this document says so.

> **target** (기계)

The place a program will run. Its word size, byte order, presence of floating-point numbers and so on are fixed, and the processor knows them before building. It may be the same as or different from the place where translation happens (the translation environment of §5.1).

2 Machine names form a **closed set**, and the following is all of it.

Table 5 — Machine names

Name	Word (bits)	Byte order	Remarks
`x86_64`	64	little-endian	floating-point numbers · vectors
`arm64`	64	little-endian	floating-point numbers · vectors
`riscv64`	64	little-endian	floating-point numbers · vectors are optional, so not in the base
`mips_be`	32	**big-endian**	floating-point numbers
`cortex_m`	32	little-endian	**no heap · no floating-point numbers** — a machine that runs alone
`cortex_m0`	32	little-endian	the same as `cortex_m` except that the multiplier width is 16 bits (⟦§5.3⟧ (4c)) — Cortex-M0 · M0+
`win64`	64	little-endian	crossing only (⟦§5.6⟧ (4))
`cheriot`	32	little-endian	**pointers are capabilities** (8 bytes, ⟦§5.3⟧ (4a)) · no heap · no floating-point numbers · crossing only (⟦§5.6⟧ (6))
`atmega328p`	16	little-endian	AVR (Arduino Uno · Nano) · alignment 1 byte · multiplier width 8 bits (⟦§5.3⟧ (4c)) · no heap · no floating-point numbers · emitted and run (⟦§5.6⟧ (8))
`atmega2560`	16	little-endian	AVR (Arduino Mega 2560) · the same values as `atmega328p` · emitted and run (⟦§5.6⟧ (8))
`atmega32u4`	16	little-endian	AVR (Arduino Leonardo · Micro) · the same values as `atmega328p` · up to building (⟦§5.6⟧ (8))
`atmega4809`	16	little-endian	AVR (Arduino Nano Every) · the same values as `atmega328p` · up to building (⟦§5.6⟧ (8))
`word16`	16	little-endian	no heap · no floating-point numbers · **stands up to translation only** (⟦§5.6⟧ (7))

2a Machine names form a **tree**. A node inherits its parent's values and states only the values it changes — `cortex_m0` changes only the multiplier width from `cortex_m`, and `atmega328p` · `atmega2560` · `atmega32u4` · `atmega4809` inherit the values of the group `avr` unchanged. **Groups** that stand only at branches of the tree (`lowent` · `riscv32` · `avr`) are not machine names and so are refused under (3). The processor shows the expanded values of each node.

3 A name not in this list matches no build and so is refused. If a name that does not match were accepted, the op would **silently drop out or be silently built wrong** — both are what this document sets out to prevent.

4 **What a machine does not have cannot be used.** The processor refuses it at translation.

Table 6 — What machines lack — refused on the spot

What is missing	Which machines	If used
a growing root (heap)	`cortex_m` · `cheriot` · `atmega328p` · `atmega2560` · `atmega32u4` · `atmega4809` · `word16`	the `heap` effect · `cap`

		`heap . region <name>` `heap` are refused (`E-HEAP-NOHOST`). `alloc` carving from a fixed window can be used ([§8.1])
64-bit floating-point numbers (`f64`)	`cortex_m` · `cheriot` · `atmega328p` · `atmega2560` · `atmega32u4` · `atmega4809` · `word16`	`f64` and floating literals without a type word are refused (`E-FLOAT-NOFLOAT`)

4-1 `f32` **exists on every target.** On a target without a floating-point unit it is computed in software, and the result is identical, to the bit, to that of a target with one. `f64` exists only on targets with the unit — it cannot be used on the targets of the table above.

4-2 The processor accepts software `f32` on every target without a floating-point unit. On the four AVR chips (`atmega328p` · `atmega2560` · `atmega32u4` · `atmega4809`) it has been run and measured to the bit; on `cortex_m` · `cheriot` · `word16` it is built but has not been run — a declaration that uses `f32` there is accepted, and translating it warns (`W-FLOAT-UNTESTED`: it may not work correctly; if a result looks wrong, tell the developer). That warning is not a rule of the language but the processor saying it has not yet measured it. A target for which software `f32` is not built yet refuses `f32` (`E-FLOAT-NOTYET`) — this edition has no such target.

> **In plain words**
>
> Why `f32` and `f64` are separated: 32-bit floating point can be computed exactly in software even on small machines, and there is much work that needs it there, such as sensor values and control coefficients. 64-

> bit is several times heavier to compute, and C compilers for small machines sometimes make `double` 32 bits — refusing is better than accepting a value that cannot be produced honestly.

5 Whether to use what a machine **has** is decided by the person building. A build chooses **which instruction sets the binary contains**, and if it contains two or more, the program chooses **once, when it starts**, and fixes the choice. It does not change during execution — if what ran differed from run to run, a measured number could not be reproduced.

Table 7 — The range of instruction sets — the build chooses

Chosen	What happens
contain nothing	every computation is done with plain code. Stands on every machine
contain one or more sets	code is emitted **assuming those instructions exist**. It does not run on a machine without them
contain them but «choose»	both are contained, and one is detected and fixed once at start. Stands on every machine, and is fast on machines that have them

5a Whichever is chosen, **the answer is the same**. Machine instructions and plain code are two implementations of the same meaning; what differs is only speed and **timing properties** — a computation that reads a table reads at a place that depends on the value, and those instructions do not. Where constant time is promised, the difference is written down (§9.2).

5c What can be contained is not only **instructions that replace a computation** — **width** (placing several independent jobs side by side in one register) is in the same range too. The rule is the same: the answer is the same (§5.6 (5a)), it cannot be contained for a machine without it (§5.6 (5b)), and the choice is made **once, at start**.

5b If the machine lacks the instruction, it cannot be contained. Asking for it to be contained makes translation refuse — it is not silently replaced with plain code. The person building shall know what will run.

4a That is, «code that runs on embedded systems too» is **something measured, not something hoped for** — a program that uses a growing root or floating-point numbers stops translating the moment that machine is chosen.

In plain words

> Why stop it at **translation** rather than at execution? Sending an allocation to a machine without a heap makes it blow up late, on that machine, at the place hardest to fix. It is better to say **in advance** that what is missing is missing — that is the opposite of what this language calls «hidden cost» in §1.3.

5d `win64` has meaning **only as far as crossing**. C source for that machine can be emitted and built, and that is as far as this document speaks; it says nothing about running on it. An op that uses something only POSIX has is refused at translation — it is better to **cry first at the place where the reason is known** than to blow up later because a header is missing.

6 `cheriot` too has meaning **only as far as crossing** — emitting C source for that machine and building it. This machine's pointers are capabilities (§5.3 (4a)), and `machine.ptr_caps` is true. The pointer of a register block is derived from the device-memory capability the execution environment gives (§7.6.1 (3)).

7 `word16` has meaning **only as far as translation**. It is not the name of a real machine but the place for measuring whether the same text stands in an implementation where the width of `usize` · `idiff` is 16 bits (§6.2 (3-1)). The processor translates and checks text for this target but neither emits

C source nor runs the text — because whether what is emitted at that width is correct has never been measured. The upper bound of a slice length is 2^{15}.

8 `atmega328p` and `atmega2560` have meaning as far as emitting C source and running it. The C source emitted for that machine is built by that chip's C compiler, and the answers from running it on a machine that emulates the chip have been measured to equal other executions of the same text. `atmega32u4` and `atmega4809` have meaning **as far as building** — there is no emulating machine, so the answers from running could not be measured. C `int` is 16 bits, so the processor emits 32-bit words as fixed-width C types. Atomic operations are emitted as read-and-write while interrupts are blocked — these chips have one core. The upper bound of a slice length is 2^{15}.

5.7 Where a unit states what it is — `package`

1 A translation unit may state **what it is** with `package <key> <value> .`. This has **the same shape** as a build declaration (`build tier t0 .` and the like) — two idioms would be a second expression too, so no new construct is made.

2 The keys form a closed set of six. Any other name is refused (`E-PKG-KEY`).

Table 8 — The keys of `package`

Key	Required	What is written
`name`	yes	the name of this unit
`version`	yes	the version number. It shall be in a form **that can be ordered**
`description`	no	a one-line description
`date`	no	a date
`authors`	no	the authors
`license`	no	the conditions of use

3 If even one key is written, `name` and `version` **shall** be present (`E-PKG-NONAME` · `E-PKG-NOVERSION`). An identity place without an identity is decoration.

4 Writing the same key twice is refused (`E-PKG-DUP`). An identity has **one** value — with two, each tool that reads this place picks one, and not all pick the same.

5 `version` shall be in **a form that can be ordered**, or it is refused (`E-PKG-VERSION`). A version number nobody can interpret cannot **order** editions, and ordering is the only reason a version number exists.

6 A shape other than `package <key> <value> .` is refused (`E-PKG-FORM`).

> **In plain words**
>
> This place exists so that tools and people can learn the identity of this unit cheaply **without reading the whole source**. But if what is read cannot be trusted, the value is as good as absent — so here alone **what is written is checked.** A key nobody reads is a key nobody checks, and an unchecked duplicate rots into a lie (§1.1).

5.8 Building differently per build — `build option` and `config`

1 A program may declare **knobs**. `build option <name> <kind> …` makes a knob, and its value is fixed where the program is built.

2 There are three kinds; anything else is refused (`E-OPT-TYPE`).

Table 9 — Kinds of knob

Kind	What it holds

`bool`	on and off
`int`	an integer
`choice`	one of the listed values

3 A knob may **depend** on another knob (`depends`). If the depending knob is a `bool` and is **on** while the knob it depends on is off, the build is refused (`E-CONFIG-DEPENDS`) — because **such a build does not exist**; without this check `depends` would be decoration.

3a If the depending knob is a `choice` or an `int`, there is no place called «on» — that knob always has a value. A mismatched dependency is then **not a nonexistent build but an unused knob**, and if the configuration nevertheless picks a value, it is reported (`W-CONFIG-DEPENDS`).

> **In plain words**
>
> Before (3a), «on» was measured only as a non-zero value, so a `choice` knob that could not pick 0 became **always on**. So **no configuration at all could exist** that switched off a knob it depended on — a knob that no configuration can switch off is not a knob (defect note 18).

4 The value of a knob is read with `config <name>`, and it is a **translation-time constant** (§6.8). Reading a name that does not exist is refused (`E-CONFIG-UNDEF`).

4a If the knob depended on **does not exist at all**, it is refused (`E-OPT-DEPENDS`).

4b If a configuration gives a knob **a value its kind cannot accept**, it is refused (`E-CONFIG-TYPE`). So is writing an unknown property (`E-CFG-UNKNOWN-PROP`).

5 A knob that is declared but **read by no code** is refused (`E-OPT-UNUSED`). Such a knob appears in the configuration, a user switches it off, and nothing happens — **something that looks like a decision but is not one**.

6 Once the values are fixed, **switched-off branches do not remain in the output.** The cost is 0.

> **Caution — Switched-off branches are still checked**
>
> In other languages' conditional compilation, code for combinations that are not switched on rots unread by anyone. That cannot happen here — what is folded away is **code emission**, not parsing and type checking. So «it does not build in that combination» does not happen.

5.9 Decided but not built: tools — verified versions · the package file · private sources · splitting a large build

1 This clause writes what the **tools** around the implementation are to do whose meaning is decided but which are not built yet. They are not words of the language, so there is no word to refuse. As in §6.9.4, they are built when a use appears and dropped if shown unnecessary. Whether to build or drop them is the owner's decision.

2 **Verified version numbers** — the kind of a version (one that breaks compatibility · one that does not) is decided not by the author but by the tool, from the difference of the unit's **visible surface** (signatures · contracts · effects · layout). Marking a version that breaks the surface as non-breaking refuses the release. When a dependency is written as a range (any compatible version), the tool picks the newest compatible one, and asks a person before moving to a breaking version. The spelling of the range is decided when built. What stands today: a version number must be orderable (§5.7 (5)), and the lock file (`lowent.lock`) pins the content hash of each dependency.

3 **The package file** (`.lowpkg`) — one file that bundles the manifest · sources · metadata · signature for release. It invents no new cryptographic format and stays a thin convention over verified bundles. What stands today: dependencies taken from a path · content hashes · signatures and trusted keys (`lowentc key` · `lowentc verify`) · mirrors.

4 **Authentication to private sources** — identifying oneself when taking a package from a private server. What stands today: the policy of where packages may come from (anywhere · this machine only · signed with an official key only) and content hashes — wherever it came from, the same hash is the same package.

5 **Splitting a large build** — dividing build information into pieces by meaning (the role of a piece · size limits · the members of a set), and a screen tool (`lowent config`) that chooses the values of the knobs (§5.8). What stands today: knobs and `config`, and `pkg.low`.

> **Note (informative)**
>
> The sources are RFC-0023 (verified versions · dependency ranges) · RFC-0032 · 0033 (sources · authentication) · RFC-0034 (`.lowpkg`) · RFC-0036 D1 to D4 (splitting a large build). The sweep of 2026-11-11 (F7) also wrote the RC11 memory orders (RFC-0018) · the decision tree of `match` (RFC-0081 MM9) · signatures · mirrors · knobs as «not built», but that was **wrong** — those five are built (the five orders of §7 · jump tables · `lowentc verify` · the mirror setting · §5.8). It was corrected on 2026-11-12. The owner decided, as for §6.9.4, to «put them in formally, state that they are dropped if they later prove unnecessary, and keep tracking them» (2026-11-12).

6 Language

1 This clause specifies the grammar and meaning of Lowent. It first specifies which pieces a source is divided into (§6.1), and then, in turn, what those pieces mean.

6.1 Lexical elements

Syntax

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

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

Static semantics

1 The processor reads a source file from the front and divides it into a sequence of `separators` and *tokens*. Tokens are of five kinds — keywords (`keyword`), names (`name`), paths (`path`, §6.1.3), literals (`literal`, §6.1.4) and punctuation (`punctuation`).

2 A `separator` divides tokens and **is discarded.** It has no other meaning.

2a Of the punctuation, a detached dot `.` closes a form and `( )` group a form (§6.1.5 · §6.1.6). An `operator` is an operator only inside an `expr` island (§6.3.2).

3 A run of letters, digits and underscores is read **as one piece, without breaking it**, and then its kind is decided. `iffy` is not something beginning with the keyword `if` but one name, and `remark` is a name, not a comment.

6.1.1 White space and comments

Syntax

```
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       ::= ? a character that is not newline ?
comment-line    ::= ? a line that is not the closing line ?
end-of-source   ::= ? the end of the source ?
```

1 The **closing line** of a `block-comment` is a line that begins, in its very first column, with the same characters as the opening line's `tag`, immediately followed by something that is not a `name-char` (§6.1.3). Every other line is a `comment-line` — lines that begin with an indented `tag`, or with a word longer than `tag`, are such lines.

Constraints

2 `rem` and `note` open a comment only when what immediately follows is not a `name-char` — `remark` and `notes` are names (§6.1 (3)).

3 `note` shall be followed by a `tag`.

4 After the `tag` that closes a `block-comment`, only `blanks` and a `newline` may follow.

5 An open `block-comment` shall meet its closing line before the source ends.

Static semantics

6 White space (space · tab · carriage return · line feed) divides tokens. It has no meaning other than dividing tokens.

7 From a place beginning with `rem` to the end of that line is a *comment*, and the processor treats it as absent. A `block-comment` is a comment too — everything from its opening line to its closing line is treated as absent.

8 Unicode characters may be used inside comments.

9 `rem` and `note` are not names. A place beginning with either of them is always a comment.

Diagnostics

Table 10 — Diagnostics of comments

Violation	Diagnostic
(3) — no tag after `note`	`E-NOTE-TERM`
(4) — other characters after the closing tag	`E-NOTE-TAIL`
(5) — the source ended without a closing line	`E-NOTE-UNTERM`

Examples

> **Example — comments**
>
> **Source**
>
> ```
> rem This whole line is a comment. Hangul may be used too.
> let n u32 42 .   rem From here to the end of the line is a comment too.
> ```

> **Example — a multi-line comment**
>
> **Source**
>
> ```
> module ex_note .
>
> note DOC
> An explanation spanning several lines.
> The processor treats it as absent.
> DOC
>
> fn f output u8 . do return 1 . end .
> ```
>
> **Output**
>
> ```
> f() = 1
> ```

Notes

> **Note (informative)**
>
> The word that opens a comment is `rem` because it is a **word**. This language has no comment marks made of symbols (`//` · `/* */`) — unifying the vocabulary as words reduces the reading rules to one.

> **Note (informative)**
>
> A `block-comment` has the same shape as a text literal (§6.1.4) but is **a comment, not a value**. The rule for the closing line is the same. A one-line comment is written with `rem`.

6.1.2 Keywords

Syntax

```
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"
```

Constraints

1 A *keyword* is a name whose meaning the language has fixed, and it cannot be used with another meaning. A spelling the same as a keyword cannot be a name.

2 The list of keywords is fixed. The production above is all of it, and it is the same as the list in annex A.

2a There is no keyword that is only reserved and not used. For every keyword there is a construct in this document that uses it.

3 A type declaration begins with its own keyword — `struct` · `enum` · `type` · `newtype`. Like other declarations (`fn` · `proc` · `actor` · `trait` · `test` · `module`), it has no word in front.

4 `def` is not a keyword. It can be an author's name. But the old heads `def struct` · `def enum` · `def type` · `def newtype` are recognised by that shape and refused.

5 `export` comes before the keyword of a declaration: `export struct rect do … end .`.

Diagnostics

Table 11 — Diagnostics of keywords

Violation	Diagnostic
(1) — a keyword was used as a name	`E-NAME-KEYWORD`
(4) — a type declaration began with `def`	`E-DEF-OLD`

Examples

Rejected example — a type declaration began with the old head `def`

Source

```
module ex_defold .

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

Diagnostic: `E-DEF-OLD`

Example — `def` can be an author's name

Source

```
module ex_defname .

struct pt do
  x u32 .
end .

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

Output

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

Notes

> **Note (informative)**
>
> Keywords are a **budget**. Adding a keyword makes the language larger, so what can be expressed with existing keywords is not given a new keyword.

> **In plain words**
>
> The word «budget» may be unfamiliar. It means this — the keywords are counted and managed, and each time one is added, it is weighed whether it is worth that much. The number of keywords in this language is kept at a size a person can hold on one screen and memorise.

6.1.3 Identifiers

Syntax

```
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 The dots of a `path` are **attached** to the characters before and after them. A dot separated by a space is not part of a `path` but punctuation (§6.1.5).

Constraints

2 A `name` cannot have the same spelling as a keyword (§6.1.2).

3 Unicode characters cannot be used in names (§5.2). The characters that make up a name are only the ASCII characters the production above specifies.

4 **A name that is made has no dot.** A name with a dot attached cannot be declared.

5 A spelling the same as the name of a core op (§6.3.3) cannot be used as a parameter or local name. The names of builtin ops are in their own namespace and so are not covered by this rule.

Static semantics

6 An *identifier* is a name the author makes. Modules · ops · types · variables have names.

7 A dot appears **only when pointing.** A `path` is **a route that points at a declared name**, and its meaning is only the three that §6.1.5 (5) specifies — the name of a module, the name of a variant, the name of a declaration belonging to a type. A dot is not part of a name but **the mark that joins names**.

8 Choosing a field inside a value is not written as a `path`. That is the `field` form (§6.2.10).

Diagnostics

Table 12 — Diagnostics of identifiers

Violation	Diagnostic
(2) — a keyword was used as a name	`E-NAME-KEYWORD`
(3) — a name contains a character that is not ASCII	`E-NAME-ASCII`
(4) — a name with a dot attached was declared	`E-NAME-DOTTED`
(5) — the name of a core op was used as a local name	`E-NAME-BUILTIN`

(8) — a field of a value was written with an attached dot (`p.x`)	`E-FIELD-GLUED`

Examples

> **Rejected example — a name with a dot attached cannot be made**
>
> **Source**
>
> ```
> module ex_dotted .
>
> fn f output u8 .
> do
>   let a.b u8 1 .
>   return 1 .
> end .
> ```
>
> Diagnostic: `E-NAME-DOTTED`

Notes

> **In plain words**
>
> This language **has no** nested namespaces. To put a hierarchy into a name, write it with underscores (`http_header_parse`). If dots were allowed in names, a reader could not tell whether `a.b` is *"field b of a"* or *"the name a.b"* — and that is semantic entropy (§1.3).

> **In plain words**
>
> Using a keyword as a name is refused **at the place of declaration** — it does not wait for a use. The diagnostic points at the name that is the problem.

> **Caution — Same names do not merely confuse; they are wrong**
>
> Some languages allow an inner name to **hide** an outer name (shadowing). A hidden name makes a person reading the source **think of a different value**, and that is semantic entropy (§1.3). So this language **forbids hiding entirely** — none of the names of core ops, the names in the module, the names of parameters, and the names alive in an outer block can be hidden (§6.4.8).

6.1.4 Literals

Syntax

```
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       ::= ? one byte that is neither a double quote nor a line feed ?
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      ::= ? one character that is not a single quote, a backslash or a line feed ?
string-char    ::= ? one byte that is not a double quote, a backslash or a line feed ?

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      ::= ? a line that is not the closing line ?
```

1 A `sign` is a sign only when it is **attached** to the number. `-5` is one `number`, and `- 5` is not. **Outside** an `expr` island (§6.3.2) a sign attached to a number is part of the value; **inside** an island `-` and `+` are operators — so `g 10 -3` in a form position is two arguments.

2 Without a point or an exponent it is read as `dec`; with a point or an exponent it is read as `float`. There shall be digits **both before and after** the point for a `float` — `.5` and `1.` are not `float`s, and that point is read as the mark that closes a form (§6.1.6).

3 A `prefix` is a prefix only when it is **attached** to the quote. If there is a space between, it is one name and one literal.

4 The **closing line** of a `text-literal` is a line that begins, in its very first column, with the same characters as the opening line's `tag`, immediately followed by something that is not a `name-char` (§6.1.3). Every other line is a `text-line` — lines that begin with an indented `tag`, and lines that begin with `tag` but continue as a longer word (`DOCS` when `tag` is `DOC`), are such lines.

Constraints

5 An underscore comes only **between** digits. It cannot come at the very start or end of a number, directly after a radix mark, or directly after another underscore.

6 After a radix mark (`0x` · `0b`) or an exponent mark there shall be at least one digit of that radix.

7 A `number` cannot be directly followed by a letter or an underscore.

8 The value of an integer literal shall be within the range of the type of the place where it is used. A floating-point literal too shall be within the range the type of its place can represent.

9 A `float` shall be written in a floating-point place and an integer literal in an integer place. The two do not convert into each other — in an `f64` place write `3.0`, not `3`.

10 A `char-literal` shall fit in **one** element as decided by the prefix. Neither an empty one nor one exceeding one element can be written.

11 The `escape`s are **only the fourteen** the production above specifies. No other character can be written after a backslash.

Table 13 — Escapes — a closed set of fourteen

Written	Value	Meaning
\\	0x5C	the backslash itself
\"	0x22	double quote
\'	0x27	single quote

`\a`	0x07	alert (bell)
`\b`	0x08	backspace
`\f`	0x0C	form feed
`\n`	0x0A	line feed
`\r`	0x0D	carriage return
`\t`	0x09	horizontal tab
`\v`	0x0B	vertical tab
`\0`	0x00	zero byte
`\xNN`	0x00–0xFF	**two** hexadecimal digits
`\uXXXX`	code point	**four** hexadecimal digits
`\UXXXXXXXX`	code point	**eight** hexadecimal digits

12 The value written by `\uXXXX` and `\UXXXXXXXX` shall be a code point. Surrogate positions (D800 to DFFF) and values above 10FFFF cannot be written.

13 The `prefixes` are only `u` and `U`. No other word can be attached to a quote. The `r` of a raw string (13a) attaches only to a double quote — there is no `r'…'`.

> **raw string** (날 문자열)
>
> A string literal that holds its body exactly as written. A backslash does not open an escape, and it ends at the closing double quote.

13a The body of a *raw string* `r"…"` **has no escapes.** A backslash is just a byte too, and the literal ends at the closing double quote. So a raw string cannot contain a double quote — if one is needed, use an ordinary string (`\"`). Its elements are bytes (`slice u8`), and it is not used together with `u` · `U`. It is the place to hold, exactly as written, text with many backslashes, such as regular expressions, paths and format strings.

14 After the `tag` that closes a `text-literal`, only `blanks` and a `newline` may follow. The dot that closes the statement is written on the **next line**.

15 On the line that opens a `text-literal`, only a `processor` and a comment may follow the `tag`. `text` shall be followed by a `tag`.

Static semantics

16 A *literal* is a value written directly in the source.

17 Integer literals are written in decimal (`dec`) · hexadecimal (`hex`) · binary (`bin`). Underscores do not count in the value (`1_000_000`).

18 **A leading 0 is not octal.** `0755` is 755. This language has no octal notation.

19 An integer literal **has no type by itself.** The place where it is used decides its type.

19-1 The precision in which the processor holds an integer literal at translation is **128 bits** — up to $2^{128}-1$ unsigned, and down to -2^{127} signed. A literal beyond 64 bits (the end of `u64` · the end of `i64`) fits only in a 128-bit place (§6.2.2 (3b)); in other places translation is refused. A literal beyond 128 bits is refused everywhere (§6.1.8 (3)).

19a An expression made only of literals — arithmetic between literals, a shift whose left side is a literal (`shl 1 n.`) — has no type either. If the place that takes that expression (a return · a binding · the other side of an arithmetic operation · an argument of a call · `set`) decides an integer type, **the expression is computed in that type.** If it overflows, or the shift count is the width or more, it stops like any other computation (§6.3.4 · §6.3.5). In a shift only the left side takes that type — the right side is the shift count.

A *floating-point literal* is a number written as a `float` — `1.5` · `1e3` · `1.5e-3`. In hexadecimal exponent notation what follows `p` is a power of 2 (`0x1p3` is 8.0).

There is no literal that writes `inf` or `nan`. Such values are obtained by computation.

True and false are written with the keywords `true` · `false`. They are of type `bool` and do not convert to and from integers (§6.2.3).

The value of a *character literal* is the code of that character, an integer.

The value of a *string literal* is a row of elements. `len` gives **the number of elements**.

An `escape` beginning with a backslash is called an *escape*. `\xNN` consumes **exactly** two hexadecimal digits, `\uXXXX` four, and `\UXXXXXXXX` eight — the number of digits does not depend on the value.

`\uXXXX` and `\UXXXXXXXX` **write a code point.** Which elements that code point is carried as is decided by the prefix — UTF-8 bytes without a prefix, UTF-16 code units with `u`, and the code point as it is with `U`.

A *prefix* decides **what the elements** of the literal are. Without a prefix they are bytes; `u` makes them UTF-16 code units and `U` code points. Character literals are the same — `u'…'` is one code unit and `U'…'` one code point.

Table 14 — What the prefix decides — `len` is always the number of elements

Written	Element	`len "한" .`	`len "ab" .`	`len "😀" .`
`"…"` (none)	byte	3	2	4
`u"…"`	UTF-16 code unit	1	2	2
`U"…"`	code point	1	2	1

A *text literal* is a string that holds several lines as they are. Its value is the `text-lines` joined with the line feeds between them. **The one line feed directly before the closing line is not part of the value** — to leave a line feed at the end of the body, put one empty line before the closing line.

In the body of a text literal **escapes are not processed.** The bytes written are the value.

The `processor` decides the elements of the value — `text TAG u` is a sequence of UTF-16 code units (`slice u16`), and `text TAG U` is a sequence of code points (`slice u32`). Without a `processor` it is a sequence of bytes (`slice u8`).

Diagnostics

Table 15 — Diagnostics of literals

Violation	Diagnostic
(5) — an underscore is somewhere other than between digits	`E-NUM-SEP`
(6) — no digit after a radix mark or exponent mark	`E-NUM-EMPTY`
(7) — a letter directly follows a number	`E-NUM-SUFFIX`
(8) — a literal is outside the range of the type of its place	`E-TYPE-WIDTH`
(9) — an integer literal was written in a floating-point place	`E-TYPE-LET`
(19-1) — an integer literal beyond 64 bits is in a place that is not 128 bits	`E-LIT-RANGE`
(10) — a character literal is empty	`E-CHAR-EMPTY`
(10) — a character literal exceeds one element	`E-CHAR-WIDTH`
(11) · (12) — an escape is outside the set or is not a code point	`E-STR-ESCAPE`
(13) — the prefix is not `u` · `U`, or `r` was attached to a single quote	`E-STR-PREFIX`

(14) — other characters after the closing tag	E-TEXTLIT-TAIL
(15) — something that is neither a processor nor a comment after the tag on the opening line	E-TEXTLIT-OPENER
(15) — no tag after `text`	E-TEXTLIT-TERM

Examples

Example — character literals

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 .
```

Output

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

Example — a code point is written, and the prefix decides the encoding

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 Control characters with names.
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 .
```

Output

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

Rejected example — what is not a code point cannot be written

Source

```
module ex_surro .

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

Diagnostic: E-STR-ESCAPE

Rejected example — there are no octal escapes

Source

```
module ex_octal .

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

Diagnostic: `E-STR-ESCAPE`

Rejected example — an escape outside the set is refused

Source

```
module ex_esc .

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

Diagnostic: `E-STR-ESCAPE`

Rejected example — there are only two prefixes — there is no `u8`

Source

```
module ex_pfx .

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

Diagnostic: `E-STR-PREFIX`

Example — a raw string — backslashes stay as they are

Source

```
module ex_raw_string .

rem `\\` in a raw string is two bytes — in an ordinary string it is one backslash
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"` is not a line feed but two bytes, a backslash and `n`
fn second output u8 .
do
  return idx r"\n" 1 . .
end .

rem A backslash directly before the closing double quote is just a byte too
```

```
fn ends_in_backslash output u8 .
do
  return idx r"a\" 1 . .
end .
```

Output

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

Rejected example — the r of a raw string does not attach to a single quote

Source

```
module ex_raw_char .

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

Diagnostic: `E-STR-PREFIX`

Example — a text literal — several lines as they are

Source

```
module ex_textlit .

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

rem Escapes are not processed in the body — `a\nb` is four bytes.
fn raw output usize .
do
  return len text RAW
a\nb
RAW
  . .
end .
```

Output

```
doc() = 17 · raw() = 4
```

Example — literals

Source

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

Example — floating-point literals

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 .
```

Output

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

Rejected example — an integer literal does not become a floating-point number

Source

```
module ex_intfloat .

fn f output f64 .
do
  let x f64 3 .     rem 3 is an integer literal — it shall be written 3.0
  return x. .
end .
```

Diagnostic: `E-TYPE-LET`

Rejected example — a literal outside the range of its type

Source

```
module ex_lit .

proc p output u8 . effects none .
do
  let x u8 300 .     rem 300 is outside the range of u8 (0 to 255)
  return 0 .
end .
```

Diagnostic: `E-TYPE-WIDTH`

Notes

In plain words

The single quote itself is written `'\''` (`'\x27'` is the same value). And `U'한'` is 54620 because `U` means a **code point**. Written without a prefix as `'한'`, that character is three bytes and does not fit in one slot, so it is refused.

In plain words

Note that `u"😀"` is 2 — this character is written as **two units** in UTF-16 (a surrogate pair). The same character is 1 with `U`. That is why the phrase *"number of characters"* is not used in this language. It always says first **in which elements it counts**.

> **In plain words**
>
> The same `\U0001F600` gives three different numbers — and **none of them is wrong.** The escape says only *“which character”*; *“in how many pieces it is carried”* is said by the prefix. Because the two are kept apart, the escape need not know the encoding.

> **In plain words**
>
> D800–DFFF are **positions** UTF-16 uses when it splits a large character into two pieces, not characters. If they were accepted, that literal would point at no character at all.

> **In plain words**
>
> In C, `\101` is octal 65, that is `A`. This language has no octal notation at all (`0755` is 755), so it does not revive octal for escapes alone — that place would then be the **only exception** in the language.

> **In plain words**
>
> `doc()` is 17: the two lines plus the one line feed between them — no line feed is attached after the last line. `raw()` being 4 is what *“exactly as written”* means: four bytes `a` · backslash · `n` · `b`, not one line feed.

> **Note (informative)**
>
> The tag position after `text` is open to someday take the name of a **processor that makes the value** (for example another encoding or a hexadecimal byte string). The processor of this edition has **only the two** prefixes above, and other names are refused. If they were passed silently, `hex"41"` would become a nameless something.

> **Caution — Without a digit before the point it is not a floating-point number**
>
> C · Python · JavaScript read `.5` as 0.5. Lowent does not — **a leading dot is the mark that closes a form.** If the closer and a literal compete for the same character, *“what is this dot”* can be known only by looking around it, and that is semantic entropy (§1.3). So always write `0.5`. Likewise `1.` is not a floating-point number either; write `1.0`.

> **Caution — An integer literal is not truncated silently**
>
> `let x u8 300 .` is not translated, because 300 is outside the range of `u8` (0 to 255). Some languages accept it by truncating it to 44, and then the 300 written in the source and the actual value 44 differ — one reads the source and still does not know the value.

> **Note (informative)**
>
> There are no octal escapes, because this language has no octal notation at all ((18)).

> **Note (informative)**
>
> The diagnostics for unclosed literals (a line or the source ending without a quote or a closing line) are gathered in §6.1.7.

6.1.5 Dots and forms

1 In the grammar of Lowent the full stop `.` is a **closing mark**. One dot closes the innermost open *form*.

2 A form is prefix notation — the name comes first and the arguments follow. **A name in a use position opens a form, and a dot closes it.** `add a. b. .` adds `a` and `b`: `a.` and `b.` are forms closed at once without arguments, that is, the values of those names, and the last dot closes `add`.

2a What is known to be a value from its characters alone — number · character · string literals and `true` · `false` · `none` — does not open a form and takes no dot.

2b A name in the following positions does not open a form and takes no dot: positions that **make** a name (the name of a declaration · the name of `let` · `var` · the name of an input · a loop variable · the name of a field), the position of a **type**, and the **word positions** fixed by core ops and statements (`cast <type> …` · `field … <member>` · `send … <handler>` · the words of pipe stages and the like). These positions are specified by the grammar of annex A.10 and the tables of annex D. The arguments of an op the user declares are **all forms** — even when a type is passed (`max_of u64. a. b. .`).

3 When a form comes inside a form, the inner form first closes with its own dot. `add a. mul b. c. . .` multiplies `b` and `c` and then adds `a`. So the start and end of a form are determined **by the characters alone** — the tree stands without knowing how many arguments an op takes. The number of arguments is compared with the declaration after the tree stands (`E-IR-ARITY`).

3a Parentheses `( … )` enclose one form. They do not change the meaning — they are written to group for readability or to make something stand out. A form inside parentheses closes with its own dot (`add a. (mul b. c. .) .`). Parentheses are not closers.

4 The number of dots is a **checksum** — if the number of open forms and the number of dots do not match, the processor issues a diagnostic. So a program with a missing dot is never silently read with a different meaning.

> **Example — prefix notation and dots**
>
> **Source**
>
> ```
> let total u32 add 1 2 . .
> let mixed u32 add 1 mul 2 3 . . .
> ```

5 A dot attached **after** a name (`a.`) is a closing dot — the same text as one separated by a space (`a .`). A dot attached **between** names does not close but means *qualification* — and qualification has **only three** meanings: the name of a module (`allocs.byte_allocator`), the name of a variant (`err.too_short`), and the name of a declaration belonging to a type (`fn pt.twice`). All three are **routes that point at a declared name**, not operations on values. A dot attached **before** a name (`.name`) is text that does not exist (`E-VOCAB-REMOVED`).

6 Looking inside a value — field access and method calls — **cannot** be written with an attached dot. That is a form, and like every other form **the name comes first**: `field` and `method` (§6.2.10 and §6.11.3). This is to give one meaning one spelling — if the same character meant four things, what is being read would be decided only by knowing where the name came from.

> **In plain words**
>
> It looks different from the familiar notation (`1 + 2`), but the rule is one — **name first, arguments after.** If infix notation is needed, it can be used inside an `expr` island, and its rules are in §6.3.4.

6.1.6 Newline is not a closer

1 A *newline* does not close a form. It is treated exactly like a space.

2 A form ends at its own closer. The closer is one dot `.`. Parentheses are not closers (§6.1.5 (3a)).

2a `do … end` are **brackets that pair with each other**. `end` closes only its own `do` and does not close forms outside the block. So each statement inside a block shall be closed by its own dot, and if `end` is met while one is still unclosed, translation is refused (`E-DOT-MISSING`).

2b **What has a block is closed by its own dot too.** Statements whose body is a block (`if` · `while` · the four loops · `match` · `case` · `region` · `borrow` · `task_group` · `pipe`) and declarations (`fn` · `proc` · `struct` · `enum` · `trait` · `actor` · `state` · `contract` · `test`) end at the dot after `end` — `if c. do … end .` · `fn f … do … end .`. A

chain continued with `else` is one statement, and there is one dot at the very end of the chain (`if a. do … end else do … end .`). If that dot is missing, translation is refused (`E-DOT-MISSING`).

2c A **value** that holds a block — `lit t do … end . · pipe xs. do … end .` — is closed by its own dot too, and the statement that uses that value is again closed by its own dot (`let x lit t do … end . .`).

2d A block cannot stand alone without the head that opens it (`E-BLOCK-NOHEAD`).

2e **There is no dot that closes nothing.** A doubled dot after a form already closed (`if c. . do`), a dot directly after the name in an op head (`fn f . input …`), and a dot in the middle of a head clause (`input a . u64 .`) do not conform (`E-CLOSER-EXTRA`). A type ends by its own grammar and so takes no dot (the dot in `input b slice u8 .` closes the clause). A head clause is closed by one dot, and the word after the dot is the word of the next clause or `do`. An `asm` clause separates its items (`reg a .`) with dots. Clause words (`vector · output · effects …`) cannot be the names of inputs (`E-NAME-CLAUSE`).

Rejected example — a dot that closes nothing

Source

```
module ex_extra_dot .

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

Diagnostic: `E-CLOSER-EXTRA`

Diagram — end closes only the block, and the full stop closes what has the block

```
if eq a. 0 . do return 1 . end .      ← the last dot closes the if statement
                └──────────┘ |        ← └┘ is the block of if, | is the full stop of if

let p lit pt do x 1 . end . .         ← the first dot closes lit, the second closes let
      └──────────────────┘ | |
```

Rejected example — a statement ending in a block is closed by its own dot too

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 .
```

Diagnostic: `E-DOT-MISSING`

Rejected example — a statement that uses a block as a value is closed by its own dot

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 .
```

Diagnostic: `E-DOT-MISSING`

Rejected example — a block without a head

Source

```
module ex_block_nohead .

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

Diagnostic: `E-BLOCK-NOHEAD`

In plain words

Why does `end` close only its own `do`? If `end` closed outer statements too, then to know what one `end` finished one would have to trace back **who held** that block. So there are only two rules: `do … end` **pair like brackets, and what a name opened a dot closes.** Statements whose body is a block and statements that use a block as a value end by the same rule (2026-10-09).

3 So the meaning is the same wherever a line is broken. There is no mark for continuing a line — no line-end continuation mark, no comma at the end of a line, no indentation rule.

4 A semicolon `;` is **not** a closer either. The closer has **one** spelling.

Example — a long form is one form even when its lines are broken

Source

```
module ex_newline .

rem A newline is white space. A form ends at its own closer `.`.
fn poly input a u64 . input b u64 . output u64 . do
  return add mul a. 2 .
             mul b. 3 . . .
end .
```

Output

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

In plain words

Wherever a line is broken, the meaning is the same. There is no separate *way* of continuing to learn — no line-end `\\` as in C, no comma at the end of a line, no indentation rule.

Note (informative)

There was a time when a newline was a closer. Before 2026-08-27 a newline closed forms. So there were three ways of continuing a line to learn, and breaking a line in the wrong place silently changed the meaning. That rule was removed. The closer is only `.`, and a newline is white space.

> **Rejected example — a semicolon is not a closer**
>
> **Source**
>
> ```
> module ex_semi .
>
> fn twice input a u64 . output u64 . do return mul a. 2 ; end .
> ```
>
> Diagnostic: `E-VOCAB-REMOVED`

> **In plain words**
>
> The semicolon was once the **third spelling** of the closer — when the processor met it, it produced a dot, literally. If three names point at the same thing, a reader has to know all three, so pure synonyms were removed (§1.3).

> **In plain words**
>
> Even with these rules, code as actually written **tends to write the dots every time**. Writing them always is easier to read than weighing, line by line, whether a newline closes. The newline closer was less a permission *"you need not write the dot"* than the grammar's acknowledgement of **the fact that a line is a statement**.

6.1.7 What is not closed

Constraints

1 What is opened shall be closed. String literals and character literals shall meet their closing quote **on the same line**, text literals and `note` blocks shall meet their closing line (§6.1.4 (4) · §6.1.1 (1)), blocks shall meet `end`, and groups).

Diagnostics

Table 16 — What is not closed, and its diagnostic

Diagnostic	What is not closed
`E-STR-UNTERM`	a string was not closed up to the end of the file
`E-STR-NEWLINE`	a **line feed** came inside a string — a string is one line (several lines are a text literal)
`E-CHAR-UNTERM` · `E-CHAR-NEWLINE`	a character literal was not closed
`E-NOTE-UNTERM`	a note block was not closed
`E-NOTE-TERM`	no tag after `note`
`E-TEXTLIT-UNTERM` · `E-TEXTLIT-TERM`	the closing tag of a text literal is missing or does not match
`E-TEXTLIT-TAIL` · `E-NOTE-TAIL`	other characters came after the tag that closes a text literal or note block — only the tag stands on the closing line
`E-TEXTLIT-OPENER`	on the line that opens a text literal, something that is not a processor came after the tag
`E-BLOCK-UNCLOSED`	a block was not closed
`E-GROUP-UNCLOSED`	a group was not closed

Notes

> **Note (informative)**
>
> These diagnostics usually occur **at the end of the source**. An opening mark cannot tell what is wrong until it is closed.

6.1.8 What a source shall have

Constraints

1 A source shall be **complete UTF-8**.

2 Every character of a source shall belong to some `token` or `separator` (§6.1). A character that belongs to none cannot appear.

3 An integer literal shall be of a size the processor can hold at translation.

4 The depth of nesting forms and blocks shall be within the limit the processor can follow. The processor does not silently truncate a program beyond the limit and translate it as a different program (§4.5).

Diagnostics

Table 17 — Diagnostics of the source

Violation	Diagnostic
(1) — there are bytes that are not UTF-8	`E-LEX-UTF8`
(2) — there is a character that belongs to no token	`E-CHAR`
(3) — an integer literal is too large	`E-LIT-RANGE`
(4) — nested too deeply	`E-NEST-DEPTH`

Examples

> **Rejected example — a character that belongs to no word**
>
> **Source**
>
> ```
> module ex_stray_char .
>
> fn f output u8 . do return 1 @ . end .
> ```
>
> Diagnostic: `E-CHAR`

Notes

> **Caution —**
>
> `..` does not exist in this language (`E-DOT-DOUBLE`). It is a spelling that once existed, and **a discarded spelling is not revived with a new meaning** — people reading old text read it with the old meaning.

> **Note (informative)**
>
> The constraints on the shape of number literals (the place of underscores · digits after a radix mark · letters attached after a number) are in §6.1.4 (5) to (7). Writing only `0x` is refused, not `0` — `0` is written `0x0`. There are no number suffixes (`3u8`) and no octal notation (`0o7`).

6.2 Types

1 Every value has exactly one *type*. A type fixes both what the value can be and what can be done with it.

2 Every type in Lowent **has a fixed size and representation.** There is no place where the processor decides at will according to the machine.

2a The processor decides the type of the value of every expression. If a value whose type the processor cannot decide is put in a place where a type is written — a bound name · an assignment · an op's arguments and return · the elements of a row · the fields of an aggregate — translation is refused (`E-TYPE-UNKNOWN`). A value whose type is not decided cannot be checked, so such a place is not let through as «correct».

> **Rejected example — binding a word that gives no value to a number**
>
> **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 .
> ```
>
> Diagnostic: `E-TYPE-UNKNOWN`

> **In plain words**
>
> For those who know C: C's `int` may differ in size from machine to machine. Lowent has no such type — `u32` is 32 bits everywhere. So there is never a need to ask *"how many bytes is it on this machine?"*.

6.2.1 Kinds of type

1 Types are divided into the following kinds.

Table 18 — Kinds of type

Kind	What it is
integer	`u8` `u16` `u32` `u64` `usize` `i8` `i16` `i32` `i64` `idiff`
floating point	`f32` `f64`
truth value	`bool`
aggregate	`struct` (a collection of named fields) · `enum` (one of several)
row	`array n t` (a row of fixed length) · `slice t` (a range that carries its length) · `matrix k d… t` (a row of several dimensions of fixed size)
answer	`result t e` (a success value or an error) · `option t` (a value or none)
pointing	`ref t` (read reference) · `mut_ref t` (write reference) · `owned t` (ownership)
capability	`cap k` (the right to produce an effect, §7.2)
flow	`task t` (the handle of a flow — `t` is the type of the value the flow will give, §10.3) · `chan t` (a channel — `t` is the type of the values it holds, §10.6)

> **Note (informative)**
>
> There is **no pointer** in this list. `ref` · `slice` · `owned` divide among them the work a pointer used to do — *"what does it point at"*, *"how many are there"* and *"who is responsible for destroying it"* are different questions, and if one pointer tries to answer all three, none of them can be checked.

6.2.2 Integer types

1 The name of an integer type is made of a sign and a width. `u` means unsigned, `i` signed, and the number after it is the width in bits.

2 Integers are represented in two's complement (§5.3).

3 `usize` and `idiff` are integers wide enough to count the **sizes and indices** of the address space. Even if they have the same width as `u64` · `i64`, they are **different types**, and they do not convert into each other automatically. `usize` is the type for lengths and indices, `idiff` the type for **the difference between two positions**.

3-1 `usize` and `idiff` have the same width, and it is **chosen by the implementation** (implementation-defined). That width is 16 bits or more and does not exceed the width of the widest integer type the implementation has. The implementation documents the width. `usize` is unsigned and `idiff` a two's-complement signed integer, so the positive numbers `idiff` holds are half the range `usize` holds.

3-2 `usize` · `idiff` **do not mix with any other integer type.** The widening table (§6.2.5 (4)) does not include them — there is no automatic conversion from `u8` to `usize`, nor from `usize` to `u64`. If only one of the two operands of an arithmetic operation or comparison is `usize` (or `idiff`), translation is refused (`E-TYPE-NOMINAL`). The same holds between `usize` and `idiff`. Number literals follow the type of their place and so are not caught by this rule.

3-3 A conversion between `usize` · `idiff` and other integer types **is written with** `cast` (§6.2.16 (2)). `cast` gives the value if it fits the target type and traps if it does not. `widen` and `narrow` are refused at translation when the source or the target is `usize` · `idiff` (`E-TYPE-NOMINAL`) — whether that conversion widens or narrows depends on the implementation's width, so a word that states a direction cannot write text that is true on every implementation. `narrow_try` · `narrow_sat` · `narrow_wrap`, whose names choose what happens when the value does not fit, accept `usize` · `idiff` as targets.

3-4 If a literal does not fit the implementation's `usize` (or `idiff`), translation is refused (`E-TYPE-WIDTH`). This is the only place **among integer conversions** where the verdict on the same text depends on the implementation — the value has a size not in that implementation's address space, so it is refused earlier than a trap during execution. Refusals of text that uses what a target lacks (§5.6 (4) · (4-2) — heap · floating point · 128-bit integers · atomic widths) are outside this sentence.

> **Caution — Text that does not rely on the width of `usize`**
>
> For the same text to stand on 16 · 32 · 64-bit implementations alike — ① write only lengths · indices · byte counts as `usize`, and keep positions inside files · times · hashes · protocol fields in types of fixed width. ② Compute where the two meet **on the wider side** (`add off. cast u64 len b. . . .`). ③ When moving a number from outside into `usize`, take it with `narrow_try`, so that the program refuses what does not fit. ④ Arithmetic on `usize` overflows at its width (§6.3), so places that multiply sizes take `chk_mul`. ⑤ A `usize` field of a struct differs in size between implementations, so it is not put in a layout that goes outside. ⑥ When taking the difference between two positions as `idiff`, `cast idiff` traps in the upper half of the `usize` range.

> **Example — between `usize` and other integers, `cast`**
>
> **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 .
```

Rejected example — `usize` and `u64` mixed as they are

Source

```
module ex_usize_mix .

fn end_of input start usize . input more u64 . output u64 . do
  return add start. more. . .
end .
```

Diagnostic: `E-TYPE-NOMINAL`

Rejected example — `usize` moved with `widen`

Source

```
module ex_usize_widen .

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

Diagnostic: `E-TYPE-NOMINAL`

3b `u128` · `i128` are integers of width 128. They stand **only on targets of implementations that have 128-bit integers** — using these words on other targets is refused at translation (`E-INT128-TARGET`). The implementation documents the targets on which they stand.

3b-1 From integers of other widths to a 128-bit place **there is no automatic widening** — write `widen u128 …` · `widen i128 …` (`E-TYPE-WIDTH`). The two operands of arithmetic · comparison · bit operations are both 128 bits or neither is. Number literals follow the type of their place — a literal beyond 64 bits fits only in a 128-bit place, and in other places translation is refused (`E-LIT-RANGE`, §6.1).

3b-2 128-bit integers stand in scalar places (locals · inputs · results · the contents of `option` · `result`) and also in struct fields and as the elements of slices · arrays · views · stacks. Their byte layout is the same as the target's C compiler gives 128-bit integers — size 16 bytes, alignment 16 bytes, least significant end first (a field written `big` reverses it). Putting them as elements of vectors (`vec` · `matrix`) or at the `extern` boundary is refused at translation (`E-INT128-PLACE`) — there is no vector unit with 128-bit integer lanes, and the ABI of the C boundary has not yet been opened.

3b-3 A 128-bit input taken from the command line is the decimal number as written. A number outside the range of the type (a negative number or 2^{128} or more for `u128`, outside $\pm 2^{127}$ for `i128`) is refused, not truncated — the same as the type range check at the entry boundary.

Example — the whole product of two 64-bit numbers in 128 bits

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 .
```

Example — a 128-bit integer in a struct field — the layout is the same as C's

Source

```
module ex_u128_field .

struct acc do
  lo u64 .
  big u128 .
end .

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

Rejected example — a 128-bit integer put at the C boundary

Source

```
module ex_u128_extern .

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

Diagnostic: `E-INT128-PLACE`

3a `usize` **is not a type that holds pointers.** Its width follows the address width, but a pointer value may be larger than an address (§5.3 (4a)) and cannot be brought back from an integer. Use `usize` for sizes · lengths · indices, and write what points as slices and references.

Table 19 — Ranges of the integer types

Type	Width	Range
`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 (depending on the target — (3b))	0 … $2^{128}-1$
`i128`	128 (depending on the target — (3b))	-2^{127} … $2^{127}-1$
`usize`/`idiff`	16 or more (implementation-defined)	follows the width — (3-1)

6.2.3 The truth-value type

1 `bool` has only two values, true and false. Its representation is 1 byte, and the only valid values are 0 and 1.

2 `bool` and integers **do not convert into each other.** An integer cannot be used in a condition position, and a `bool` cannot be counted as a number.

> **Rejected example — an integer cannot be used in a condition position**
>
> **Source**
>
> ```
> module ex_cond .
>
> proc p input n u32 . output u8 . effects none .
> do
>   if n. do return 1 . end .     rem `n` is not a bool
>   return 0 .
> end .
> ```
>
> Diagnostic: `E-TYPE-COND`

> **Caution — 0 is not false**
>
> C-family languages treat 0 as false and everything else as true. Lowent does not. Using an integer in a condition, as in `if n. do`, is refused at translation — **what is being asked** shall be written, as in `if gt n. 0 . do`. If what is asked is not written in the source, the reader has to guess it.

6.2.4 Floating-point types

1 `f32` and `f64` are the IEEE 754 binary 32-bit and 64-bit formats respectively.

2 Floating-point operations happen **at the width of their operands**. Operations between `f32`s are computed as `f32`, and the processor does not secretly compute in wider precision.

2a One operation is **one rounding**. The processor does not merge two operations into one rounding — it does not turn a multiply-and-add (`add mul a. b. . c.`) into one fused multiply-add instruction (FMA). Rounding is the single mode «to nearest, ties to even», and a program cannot change it. Code emitted as C carries this promise in its source too — it writes directives so that the C compiler building that code does not merge them itself.

3 Integers and floating-point numbers do not convert into each other automatically. To convert, it shall be written explicitly.

4 If the result of an operation is NaN, that it is NaN is fixed. The sign bit and the remaining bits of that NaN are a **free choice** (FC-0004).

> **Free choice — FC-0004 — the sign of a NaN produced by an operation**
>
> When the result of a floating-point operation (including per-lane operations and `reduce_*`) is NaN, its sign bit and remaining bits are chosen by the processor. That the result is NaN, and the value of every result that is not NaN, are the same either way. The difference shows when the value is printed (`nan` and `-nan`) and when it is read as bits. IEEE 754 does not fix this place either, and machines and C compilers give different signs where two NaNs meet or at `0.0 / 0.0`. Nailing it to one side would add a correcting instruction to every operation. To know whether a value is NaN, compare it with itself, as in `ne x. x.` — do not ask its sign.

6.2.5 Conversions between types

1 Lowent has **no implicit conversion that loses a value.** A narrowing place is always written in the source. The only conversion the processor does by itself is the widening below; the word that writes any other conversion is `cast` (§6.2.16).

2 A conversion that loses no value is called *widening*. Widening does not change the value, so the processor may do it automatically — a `u8` value may be passed as it is to a place that takes `u32`.

2a But **widening may also be written in the source.** `widen u64 x. .` widens `x` to `u64`. Even though the result is the same, it is used when the place where the width changes should be visible.

2b Widening to 128-bit integers (`u128` · `i128`) **is not automatic** — it shall always be written with `widen` (§6.2.2 (3b-1)).

In plain words

In short — **widening is automatic, narrowing is by hand.** Widening keeps the value, so there is no room for mistakes, but narrowing can make a value disappear, so the author shall leave in the source that they know it.

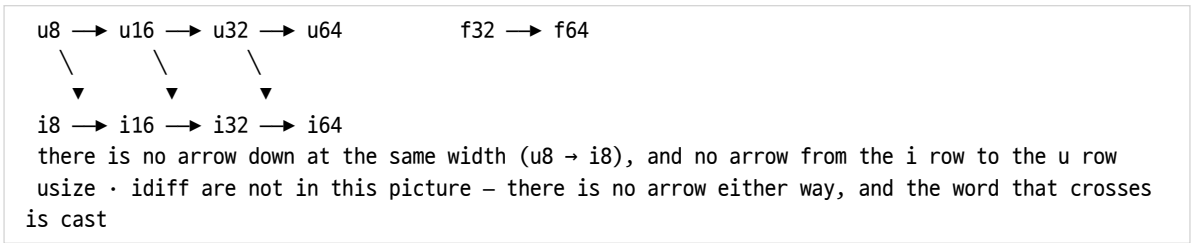
3 A conversion that may lose a value is called *narrowing* and is written with `narrow`. Narrowing **traps** if the value does not fit the target type. It is not truncated silently.

4 The relations in which widening is allowed are as follows. A combination not listed here is not widening, and writing it is refused at translation (`E-WIDEN-KIND` · `E-WIDEN-SIGN` · `E-WIDEN-NARROW`).

Table 20 — Relations in which widening is allowed

Relation	Explanation
`uN` ⟶ `uM` (N ≤ M)	an unsigned integer to a wider unsigned integer
`iN` ⟶ `iM` (N ≤ M)	a signed integer to a wider signed integer
`uN` ⟶ `iM` (N < M)	an unsigned integer to a **wider** signed integer
`iN` ⟶ `uM`	**not allowed** — negative numbers cannot be represented
`f32` ⟶ `f64`	a floating-point number to a wider floating-point type
integer ↔ floating point	**not widening** — when the kinds differ, it shall be converted explicitly
`usize` · `idiff` ↔ other integers	**not widening** — the width depends on the implementation. Write it with `cast` (⟦§6.2⟧ (3-3))

Diagram — widening — following the arrows, the value fits as it is



Example — widening and narrowing

Source

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

> **Caution — A negative number cannot be widened to an unsigned type**
>
> Widening −1 of `i8` to `u64` turns the value into a very large number. That is not widening but **becoming a different value**, so this language does not accept it as widening.

6.2.6 Rows — arrays and slices

1 `array n t` is a row of exactly `n` values of type `t`. The length is part of the type and is fixed at translation time.

1a Write **the length `n` first, as an integer literal, and the element type after it** — the same order as `vec n t` · `matrix k d… t`. Since the element type comes at the very end, the type knows its own end. Written the other way round (`array t n` — a type in the length position), it is refused (`E-TYPE-ORDER`; `lowentc --fmt` moves it to the new order). If the length is not a literal, it is refused (`E-TYPE-ARRAY`).

1b An op input `input x array n t .` means that it takes a `slice t` of length exactly `n`, and that length is **checked at entry**, like `requires eq len x. . n. . ..`

1c An op **can return** an array — `output array n t .`, or with a name, `output r array n t .` (§6.4.1 (3e)). An array has no way of being returned as a value, so **the caller provides the place**:

- The caller binds the result to a name — `var x array n t f a … .` (`let` is the same). The bytes in that name's frame are the place of the result,

and the callee builds directly there (it does not copy).

- Inside an op that returns the same kind of array, `return g … .` passes on its own place as it is. But if an argument given to `g` can see the same bytes

as that place (the named result itself, or a name derived from it — a local whose initialiser or `set` right side mentioned it), the processor **takes it into a temporary place and copies once** — while `g` writes, the reading side does not see new values, so the answer is the same as when they do not overlap. It passes the place on directly only when they are seen not to overlap (when the arguments mention no such name). A `return` outside a block (`guard … else return …`) has no place to insert a statement, so this rule does not stand there — the `E-RESULT-PLACE` below applies as it is.

- Calling an op that returns an array in any other place (the middle of an expression · an argument of another call · the source of `copy`) has no place, so

translation is refused (`E-RESULT-PLACE`). Bind it to a name first.

- A named result starts at 0 (0 in every slot). `return v. .` of an unnamed result copies `v` into the place and returns. This shall be a `return`

inside a block — outside a block, as in `guard … else return v. . .`, it is refused (`E-RESULT-PLACE`). Such an op names its result.

- The elements are of number types and the length is an integer literal (anything else is `E-TYPE-ARRAY`).

1d **Arrays inside arrays** — the element type may itself be an array. `array n array m t` is a row of `n` rows of length `m` laid out row-major without gaps (the same layout as C's `t a[n][m]`). Since the element type comes at the very end, no parentheses are needed ((1a)).

- `idx a. i. .` is a `slice t` (length `m`, no copy) that **sees** the bytes of row `i` — it follows the same rules as the sequence of rows of (9a). If the outer

one is `mut`, the slots of that row can be written too, and there is no `set idx a. i. . <row> .` that replaces a whole row (`E-TYPE-SET`).

- An op input `input x array n array m t .` takes a slice of rows `slice array m t`, and that the number of rows is `n` is **checked at entry** ((1b)).

A sequence of rows, and a row view over bytes (§8.7 (7a)), can be passed as they are.

- As a struct field it holds the bytes of `n × m` elements **inside the record** (the same as the array field of §6.2.7 (1a) — copying the struct copies them too).
- A value is built with `lit array n array m t <row>… [_] .`. Each row is `lit array m t …`, and a final `_` fills the remaining rows with 0 ((6a)).
- The elements are sized numbers and `bool`. There is no limit on the number of levels — `array 2 array 3 array 4 u8` is 2 planes of 3 rows of length 4 (C's `u8 a[2][3][4]`), and `idx a. i. .` is a slice of rows `slice array 4 u8` that **sees** plane `i`. Each `idx` peels off one level. The rows of a sequence of rows · of a row view can also be arrays inside arrays (`slice array 3 array 4 u8 · view_array array 3 array 4 u8 b.`).
- The depth of levels is part of the type — passing or binding a slice of rows of a different depth is refused (`E-TYPE-ARG · E-TYPE-LET`). A slice of rows does not fit a place for a slice of elements either (`idx` gives a row instead of a number).
- An op **can return** an array inside an array too — `output r array n array m t .`. As in (1c), the caller provides the place (`var x array n array m t f … .`),

and a named result starts as rows of 0.

> **Example — arrays inside arrays**
>
> **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 .
> ```

> **Example — three levels — each idx sees the level below**
>
> **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 .
> ```

Rejected example — a slice of rows of a different depth

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 .
```

Diagnostic: `E-TYPE-ARG`

Example — an op that returns an array — the caller provides the place

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 .
```

Rejected example — calling an op that returns an array in the middle of an expression — there is no place for the result

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 .
```

Diagnostic: `E-RESULT-PLACE`

2 `slice t` points at a range of values of type `t` laid out contiguously, and **carries a start and a length together.**

3 The length of a slice is read with `len`. `len s. .` is the number of elements of `s`, and its type is `usize` (§6.2 (3a)). The numbers given by `bitset_count` · `stack_count` and `capacity` are `usize` too. `len` used on the capability `cap args` is the number of arguments received.

3′ Indices and bounds (the `i` of `idx`, the start and end of `subslice`, the index of atomic operations, the `capacity` of `alloc_bytes`) **are** `usize`. Putting a value of another integer type in that place is refused at translation (`E-TYPE-NOMINAL`). Integer literals are written as they are. Where they meet numbers that are not lengths, write the conversion — `cast u64 n. .` moves a `usize` to `u64`, `cast usize x. .` moves a `u64` to `usize`, and either way it traps if the value does not fit (§6.2 (3-3)).

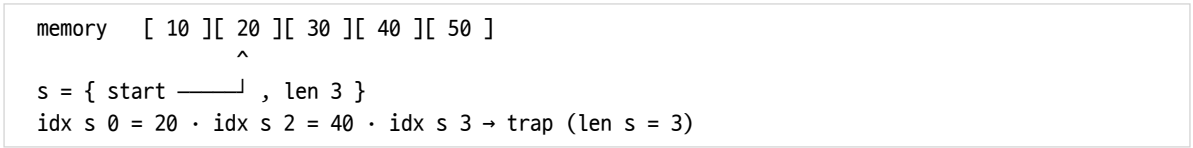
3″ A value written to an element shall be of the slice's **element type** — in `set idx s. i. . v .`, if `v` does not widen to the element type, translation is refused (`E-TYPE-WIDTH` · `E-TYPE-SIGN` · `E-TYPE-NOMINAL`). `slice usize` and `slice u64` are **different types** even on an implementation where their elements have the same width (`E-TYPE-NOMINAL`) — declare a table that holds only positions as `slice usize`, and keep a table that holds numbers of several meanings at a fixed width and write `cast` where it is read and written.

3‴ `usize` counts **what belongs to one piece**: its length · an index within it · the number one operation read, wrote or processed in that piece. Such numbers cannot exceed the length of that piece, so they fit `usize` on every implementation. **Running totals** accumulated over many operations, and sizes · positions outside memory (the size and position of a file · the sum of bytes passed through a stream · the bit length of a whole message), can be large regardless of the address width, so they are **not** `usize` — keep them in fixed-width integers (`u64`). Hash values · checksums · random numbers are not lengths: keep them at the width of the value.

4 `idx s. i. .` is the `i`-th element of `s`. The number of the first element is 0.

5 If `i` is not less than `len s. .`, it **traps**. If the processor proves that the condition is always true, the check disappears (§6.4.6).

Diagram — a slice carries a start and a length together



Example — a slice

Source

```
module ex_slice .

rem A slice carries a start and a length together.
export fn head input data slice u8 . output u8 .
  requires ge len data. . 1 . .
do
  return idx data. 0 . .
end .
```

Rejected example — the element type was written before the length

Source

```
module ex_array_order .
```

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

Diagnostic: E-TYPE-ORDER

6 **Element-list literals** — `lit array n t v₁ … vₖ .` is an `array n t` value, and `lit slice t v₁ … vₖ .` a `slice t` value of `k` elements. A list is **a form closed by its own full stop**: at the end of a statement the list's dot and the statement's dot come one after another (`let t lit array 4 u8 1 2 3 4 . .`); in the middle it is closed with one dot and the next operand follows. Since `lit` shows the type of the value, the binding statement does not write the type separately.

6a The number of array elements shall equal `n`. If there are fewer, a _ shall be written at the **end** of the list to say «the remaining slots are 0»; if there are more, or _ is somewhere other than the end, or a slice list has _, it is refused (E-LIT-COUNT). Slots are not filled silently.

6b Each element shall fit `t` — a negative number does not fit an unsigned `t` either (E-TYPE-WIDTH). `bool` elements are only `true` · `false`.

6c **Filling chosen slots** — `lit array n t do <index> <value> . … [_ <value> .] end .` puts each value into the slot whose index is written. The indices are integer literals from 0 to `n - 1`, each written once. `_ <value> .` comes only at the end and means «every slot whose index is not written». Every slot shall be decided — if slots remain and there is no `_ <value> .`, or _ has no slot to fill, it is refused (E-LIT-COUNT). If an index is not a constant, is outside the length, appears twice, or _ is somewhere other than the end, it is refused (E-LIT-INDEX). Values may be expressions and are computed in the order written; the value of _ is computed **once**. A list written as a block is a form too and is closed by its own dot — in `let t lit array 8 u8 do 2 5 . _ 0 . end . .` the first dot closes the list and the second closes `let`. It is not mixed with the form that lays out elements in a row

6d **SIMD values** — `lit vec n t v₁ … [_] .` is a `vec n t` value. The element rules are the same as for array lists; in addition `t` is a sized number (not `bool`), and `n` is a power of 2 from 1 to 16 (E-LIT-COUNT). The type of the value shall agree with the `vec` type of the binding place in number of lanes, element kind and width (E-TYPE-LANES · E-TYPE-WIDTH). Since it ends with placing values into lanes, the place rules of (7) do not concern it.

7 **Where a list is placed and how long it lives** is decided by the processor at translation as one of three. The author does not choose. ⓐ A list bound with `var`, and a `let` list containing elements computed during execution, have **one slot per declaration site** in that op's frame. That slot is filled anew each time the declaration is executed (slots not written are 0) and lives until **the block that holds the declaration** ends — a declaration inside a loop is filled again on every iteration, and recursive calls each have their own slot. ⓑ A list whose elements are all translation-time constants and which is bound with `let`, or used only for reading without a receiving place, is **read-only** bytes fixed in the program. It lives to the end of the program, so it may be held anywhere. Binding it to a `mut` name (putting it back with `set` is the same), or passing the name bound with `let` to a `mut` · `owned` · `mut_ref` place, is refused (E-TYPE-ARGMUT). **Whether two lists with the same contents (string literals too) are the same place is a free choice** (§4.3 · FC-0003) — the processor may merge them into one copy or keep them apart. So a `same_slice` for which it is visible at translation that both sides came from such literals is refused (E-LIT-IDENTITY — the literal itself · a name bound to it with `let` · its `subslice`). Cases not visible across calls are not refused, but a program that relies on that answer does not conform. ⓒ A list used without a receiving name that contains elements computed during execution, or is passed directly to a `mut` · `owned` · `mut_ref` place, is a temporary in the frame that lives **until that statement ends**. The receiving op may rewrite those slots.

Free choice — FC-0003 — whether read-only lists with the same contents are the same place

> The processor may **merge** two read-only lists with the same contents (a string literal is a list too) into one copy at the same place, or keep them **apart**. Those two are the only range of choice — the contents and length of the list, and that it lives to the end of the program, are the same either way. There is one place where the difference shows — `same_slice`, which asks whether two lists are the same place. So if it is visible at translation that both sides came from such literals, the question is refused (`E-LIT-IDENTITY`), and where it is not visible, a program that relies on the answer does not conform. To know whether the contents are the same, compare the contents — do not ask about the place.

7a If a value that sees a slot of ⓐ · ⓒ is put in a place that lasts longer than its lifetime — `return`, a name declared outside the declaring block (including parameters and their fields · elements), a message sent to an actor declared outside the declaring block — it is refused (`E-LIT-ESCAPE`). It may be handed to an actor declared in the same block or further in, and the values that actor gives afterwards receive the same lifetime. Putting a temporary of ⓒ into a name is refused too — bind the list with `var` and use it as ⓐ. The flow is followed across calls: the result of a call carries only what belongs to the inputs that flow into the result in that op's body, and a call that returns a scalar carries nothing out (the same distinction as region escape in §8).

7a-1 **An unnamed struct value** (the result of a call · `lit`) ends at the end of its statement. Putting a value that sees an array field of that value into a name · `return` · an outer place is refused (`E-VIEW-TEMP`) — once the statement ends, those bytes belong to the next value made. Only reading on the spot (taking numbers out with `len` · `idx`) conforms. To see a field for long, bind the struct to a name first (`let b T … .` and then `field b. <field> .`).

7b A list bound with `var` is a **writable** slice — slots are written with `set idx buf. i. . v. .`. `set buf …`, which rebinds that name to other storage, is refused (`E-ARRAY-SET`). Each element follows the width rule of (6b), and elements computed during execution shall agree with the element type (`E-TYPE-SET` · `E-TYPE-WIDTH`). A slot reached through `idx` and `subslice` is the same storage too — in `set idx subslice buf. a. b. . i. . v. .`, if `buf` is not a writable name it is refused (`E-TYPE-MUT`). A slice that cannot be written cannot be rebound to a `mut slice` name — `let w mut slice u32 g. .` and `let w mut slice u32 subslice g. a. b. . .` are refused if `g` is not a writable name (`E-TYPE-MUT`) — and so is putting it back into a `var` of type `mut slice` with `set w. g. .`. The new name points at the same storage, so writing through it is writing to `g`. A view `view_array <type> g.` does not copy either, so it is the same — binding it to a `mut` name, passing it to a `mut` input, or doing `set` · `copy` · `store` through it requires `g` to be a writable name (`E-TYPE-MUT` · `E-TYPE-ARGMUT`). The same holds when the receiving side of `copy` and `store` is a `subslice`. A `var` of a slice (or view) type without `mut` can be rebound to another slice but is not a name that writes slots — writing a slot through that name is `E-TYPE-MUT`, and passing it to a `mut` · `owned` input is `E-TYPE-ARGMUT` (whatever it holds — a read input or a string literal alike).

7b-1 **Copying contents** — `copy <receiver> <giver> .` copies the contents of the giving slice into the receiver. The receiver shall be writable (`E-TYPE-MUT`), the two element types shall be the same (`E-TYPE-SET`), and the two lengths shall be the same — if they are known to differ at translation it is refused (`E-COPY-LEN`), and if they differ during execution it stops. Even if two pieces of the same row overlap, it copies correctly (like a move whose ends overlap). Since it writes to the receiver, an `fn` that copies into a parameter is not pure (`E-EFFECT-PURITY`).

7b-2 **Swapping** — `swap <slice> <i> <j> .` exchanges the values of two slots. Since it writes slots, that slice shall be writable (`E-TYPE-MUT` — a `let` list · a string · a parameter that is not `mut` · an array built at translation cannot be written).

7c The total size of all slots of ⓐ · ⓒ in one op has a limit the processor states, and exceeding it is refused (`E-FRAME-SIZE` — it is not silently moved to the heap). This processor's limit is 65,536 bytes on a machine with an operating system · 4,096 bytes on a machine without one. If an op with such slots can reach itself (even through other ops), it is refused (`E-FRAME-RECURSIVE`) — frames multiply by the

recursion depth, and that depth is not known at translation, so it cannot be shown that the stack does not run out. A list inside recursion is taken from an allocator ((7d)). If, following the call relation, the sum of ⓐ · © slots alive at the same time on the deepest path exceeds the budget below, translation is refused (`E-STACK-BUDGET`) — it is measured per entry (an op nobody calls). Ops that run as tasks (`spawn <op>` · the handler of `spawn send`) are measured separately against the green-thread budget. During execution, too, the sum of ⓐ · © slots and the struct slots of (7c-1) **alive at the same time** on one call chain has a budget the processor states, and it **stops** rather than enter an op that would exceed it — the stack does not overflow and break silently. This processor's budget is 4 MiB on a machine with an operating system · 64 KiB per green thread · 16 KiB on a machine without an operating system, and every back end counts with the same numbers and stops at the same place.

7c-1 **The place of a struct value is its op's frame.** For each op the processor decides at translation the number of frame slots that struct value `lit`s and calls returning a struct will use — a value at a place for which there are not enough slots lives in the pool of §8.1 (3a). The share of the slots (8 bytes per field · the bytes of an array field) counts towards the budget of (7c). When the frame ends, the value ends too, and a returned value is **copied** into the caller's place. Recursion is not refused for frame slots — frames pile up by the depth, and the budget of (7c) stops it during execution. If, on a path without recursion, struct values push the deepest path beyond the budget, translation is refused (`E-STACK-BUDGET`). An op that takes or makes references can write to someone else's place, so it does not use frame slots (its values live in the pool).

7d **A list taken from an allocator** — `var <name> use <allocator> mut slice t lit array n t … else <statement>` (`lit slice` is the same) asks that allocator for the byte count of the list (`send <allocator> reserve <bytes>`); if it receives them, it fills those bytes as in (6)–(6c) and binds them to the name; if not, it goes to `else` (§6.5.1 (5)). To carry the failure along as an `option`, write the whole type — `let <name> use <allocator> option mut slice t lit … .`. If there is neither `else` nor `option`, or it is `lit vec`, it is refused (`E-LIT-USING`). If the element of the binding type differs from the element type of the list, it is refused (`E-BIND-ELSE`). The bytes belong to the allocator, so unless (7e) applies they do not receive the declaring-block lifetime of (7a) but follow the allocator's lifetime: if the allocator received a slot of ⓐ as its backing (`send <allocator> init buf`), the bytes that allocator gives receive that slot's lifetime too. A struct value is built in allocator bytes with the same spelling: `var <name> use <allocator>. s lit s do … end . else <statement> .` asks for `size_of s .` bytes; if it receives them, it fills them with 0, lays the layout of s over them (the same representation as `view` of §8.7.2) and writes the fields given. The binding type shall be s (`E-BIND-ELSE`). **A slice field of numbers** is placed in the allocator bytes as two words (address, length) — the same as C's `struct { const T *p; size_t n; }`, aligned to 8 bytes. This layout exists only for structs built in an allocator: laying such a struct over someone else's bytes with `view`, or asking its size with `size_of`, is still refused (bytes must not become addresses). If a value given to a slice field sees a place in a frame, the struct receives that place's lifetime too — even if kept with `keep`, carrying it longer than that place is refused (`E-LIT-ESCAPE`). Structs with `owned` · actor · capability fields, slice fields whose elements are not numbers, or inner structs without a layout are not built (`E-LIT-UNBUILT`) — those bytes would have to end together with their fields.

7e **Given back at the end of the block** — if the allocator provides `freeing_allocator` (§8.13 (6)), bytes received through (7d) are given back to the allocator when leaving the block that declared the name (`send <allocator> release <the piece received>`). This holds on reaching the end of the block · on `return` · on `break` · `continue` inside that block. If several were received in one block, they are given back in reverse order of declaration, and what was not received (what went to `else`) has nothing to give back. The answer of `release` is not used — whether to take it back is the allocator's policy. So these bytes receive the declaring-block lifetime of (7a): carrying them outside the block is refused (`E-LIT-ESCAPE`). Bytes of an allocator that is not a `freeing_allocator` stay with the allocator. When stopping with `panic`, they are not given back — stopping hands that op over to the recovery unit. The author

can give them back earlier with `drop <name> .`. Then they are not given back again at the end of the block. `drop` is accepted only in the block that declared the name (in an inner block it is `E-OWN-JOIN`), and using the name after it is refused (`E-OWN-MOVED`).

7f **Kept** — `var <name> keep <allocator> … else <statement>` switches off (7e): the bytes are not given back on leaving the block, so they follow the allocator's lifetime (§6.2.6 (7d)) and can be carried outside the block — by `return` too. Giving them back is the author's job (`send <allocator> release <piece>`). Otherwise they all go back together when the allocator ends. `keep <allocator>` stands in the place of `use <allocator>` (directly after the name) and is used only on bindings that take a list or struct literal (`E-USING-FORM`).

8 An array list bound with `let` is a **slice that sees** those bytes. The list decides the length.

9 The elements of a list may be structs with a byte layout (all fields sized numbers — structs on which `view` of §8.7.2 can stand). An element is a copy of the value given, and the list is a `slice s` with a view like `view_array` laid over those bytes. A field of an element is written with `set field <list>. <position> <field> . v. .`. Place · lifetime · `_` · filling chosen slots · the allocator clause are the same as for number elements. If the value in an element position is a number or a truth value it is refused with `E-TYPE-FIELD`; if it is another struct, with `E-TYPE-STRUCT`. Lists this processor has not yet built — structs without a byte layout · structs whose fields are structs · filling chosen rows (`do … end`) — are refused saying what is not yet built (`E-LIT-UNBUILT`). Array fields inside structs are specified by §6.2.7 (1a).

9-1 A struct whose fields are only sized numbers · `bool` and **slices of numbers** can also be an element of a list. Inside an element a slice field is placed as two words (the place it points at, the length) — the same layout as a struct built in allocator bytes ((7d)). The fields are written at execution, so such a list stands in a frame place of ⓐ. A `slice s` input taking the same elements, and an array result `output r array n s .`, use this layout too. A slice field only points at someone else's bytes, so copying an element does not copy the bytes it points at.

9a **A list of rows** — the type of a list `lit slice array m t <row>… .` whose elements are fixed-length arrays (`array m t`) is `slice array m t`. Like other slice lists ((6)) the number of rows is not written and there is no `_` ((6a)) — a row of 0 is written `lit array m t _ .`. `lit array n array m t <row>… [_] .`, with the number of rows written, is the value of an array inside an array ((1d)). Over existing bytes a row view of the same type is laid with `view_array array m t b.` (§8.7 (7a)). `len` is the number of rows, and `idx g. r. .` is a `slice t` (length m, no copy) that **sees** the bytes of row r — if the outer one is `mut`, the slots of that row can be written too. Row values are copied slot by slot, and if the shape of a row literal differs from `array m t` it is refused (`E-LIT-COUNT`). There is no `set idx g. r. . <row> .` that replaces a whole row (`E-TYPE-SET`) — write the slots, or copy with `copy`. The rows (and struct elements) of a list in a frame see that frame's bytes, so carrying them outside the block is `E-LIT-ESCAPE`.

Example — element-list literals

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 An array list is closed by its own dot — at the end of a statement there are two dots.
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 .
```

Example — a var array filled by a loop

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 .
```

Example — filling chosen slots

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 .
```

Rejected example — a slot index written twice

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 .
```

Diagnostic: `E-LIT-INDEX`

Rejected example — returning a var array from inside a block

Source

```
module ex_frame_escape .

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

Diagnostic: `E-LIT-ESCAPE`

Rejected example — fewer elements than the length, and no _ at the end

Source

```
module ex_list_short .

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

Diagnostic: `E-LIT-COUNT`

In plain words

In a language with only pointers, a person has to know separately *"how many are there where this pointer points"*. That knowledge is not written in the source, so it is easily wrong, and when it is wrong someone else's memory is read. A slice removes the problem by **putting that knowledge inside the value**.

6.2.6.1 Rows of several dimensions — `matrix`

1 `matrix k d1 … dk t` is a row of `k` dimensions holding `d1 × … × dk` values of type `t`. The number of dimensions and the sizes are part of the type and are fixed at translation. If even one of the number of dimensions or the sizes differs, it is a different type — `matrix 2 3 4 u32` differs from `matrix 2 4 3 u32` and from `array 12 u32`.

1a Write the number of dimensions first, then as many sizes as there are dimensions, and the element type at the very end (the same order as §6.2.6 (1a)). The number of dimensions is from 2 to 8 and the sizes are 1 or more, all plain decimal literals (in an op input a size may be the name of a translation-time value input — (3a)). The old order (`matrix t k d…` — element type first) is refused (`E-TYPE-ORDER`). The number of elements (the product of the sizes) cannot exceed 2^32. `matrix` or `array` cannot be written as the element type. Violations are refused (`E-MATRIX-TYPE`).

Note (informative)

Since the number of dimensions is written first, where the sizes end is known from the characters alone (a size may also be the name of an input). Even when elements follow (`lit matrix 2 2 3 u32 1 2 3 4 5 6 .`), where the sizes end and the elements begin is separated by the characters alone.

2 **Layout.** The elements are laid out contiguously in one block, row-major — the last index runs fastest. Size and alignment are the same as `array n t` (`n` is the product of the sizes), and no header is added.

Note (informative)

This layout is the same as C's `t a[d1]…[dk]`. Passing `flat m. .` to an `extern` op with a `slice t` input, C receives a pointer and the number of elements, and reading that pointer as `t (*)[d2]…[dk]` points at the same slots.

3 A `matrix` type is written **in places that make a name** — the type of `let` · `var`, an op's `input` and `output`, a struct field. It cannot be written as the element of another type, as a type argument or in the place of an alias (`E-MATRIX-USE`).

3a Only in an op's `input`, the name of a translation-time value input of that op can be written in a size position — `input comptime r u64 . input comptime c u64 . input m matrix 2 r c u32 .`. The caller writes a number for that input (`sum 3 4 a. .`), and the shape of the argument shall be the same as the shape read with those numbers (`E-MATRIX-SHAPE`). The sizes of `let` · `var` · `output` · struct fields and of `lit matrix` shall be numbers (`E-MATRIX-TYPE`).

> **Note (informative)**
>
> An input that takes its sizes by name gets no entry contract on its length. What can come in that place is only a `matrix` whose shape the caller compared, and reading and writing elements receive the checks of (5) unchanged.

4 **Values.** `lit matrix k d1 … dk t <element>… .` lists the elements in row-major order. The number of elements shall equal the product of the sizes; if there are fewer, write _ at the end to fill the rest with 0. If a binding writes both a type and a literal, the two shapes shall be the same. Violations are refused (`E-MATRIX-SHAPE`). In a struct literal, the value of a `matrix` field is a `lit matrix` of the same shape as that field.

5 **Reading and writing.** `idx m. i1 … ik .` takes one index per dimension. If the number of indices differs from the number of dimensions, it is refused (`E-MATRIX-INDEX`) — an `idx` with fewer indices does not mean a row (a row is the `row` of (7a)). Each index is checked **against the size of its own dimension**: if any one of them is that size or more, execution stops. Index expressions are computed once each, in the order written. `set idx m. i1 … ik . v. .` writes that slot, and `m` shall be a writable name (`E-TYPE-MUT`).

6 `dim m. j .` is the size of dimension `j` (counting from 0). `j` shall be a plain decimal literal less than the number of dimensions (`E-MATRIX-INDEX`). Its value is a translation-time constant.

7 `flat m. .` is a `slice t` that sees the same elements as one row. It does not copy, and its length is the number of elements.

7a `row m. i1 … i(k-1) .` is the **row** that leaves one last dimension — one index for each dimension except the last, and its value is a `slice t` that sees the elements of that row. It does not copy, and its length is the size of the last dimension. If the number of indices differs it is refused (`E-MATRIX-INDEX`), and each index is checked against the size of its own dimension as in (5). If `m` is a writable name, that row can be bound to a `mut slice t` name and written; otherwise it cannot be bound that way (`E-TYPE-MUT` · `E-IMMUTABLE` if it is a field of a struct bound with `let`). There is no view that leaves two or more dimensions (a plane).

7b In the `m` position of (5) · (6) · (7) · (7a), not only a name written as `matrix` stands but also `field x. g .` that points at a `matrix` field — `idx field b. g . r. c. .`.

8 `len m. .` and `for x m. do … end .` are refused (`E-MATRIX-USE`). The size of one dimension is written with `dim`, and going over all elements with `for x flat m. . do … end .`.

9 The places where a `matrix` name can be written whole are the first argument of `idx` · `row` · `dim` · `flat`, an argument of an op with an input of the same shape, and the `return` of an op with a result of the same shape. Other places are refused (`E-MATRIX-USE`). Arguments and inputs, and the type of a binding and the result of the op called, shall agree in both number of dimensions and sizes (`E-MATRIX-SHAPE`).

9a A call of an op whose result is a `matrix` is placed in only two places — the value of a binding that makes a `matrix` name of the same shape, and the value of the `return` of an op with a result of the same shape. The `return` of an op whose result is a `matrix` gives a `matrix` name of that shape or a call of an op that gives that shape. Violations, including binding it to an `array` name or returning it from an op that gives an `array`, are refused (`E-MATRIX-SHAPE`).

10 This type specifies only **places**. It does not specify computations such as matrix multiplication.

> **Example — fill a table of 3 rows and 4 columns and add up one row**

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 .
```

Example — a table as a struct field, one row, an op that takes any size

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 .
```

Rejected example — two dimensions, but only one index given

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 .
```

Diagnostic: E-MATRIX-INDEX

Rejected example — passing 4 rows of 3 columns to an op that takes 3 rows of 4 columns

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 .
```

Diagnostic: E-MATRIX-SHAPE

6.2.7 Aggregates — struct and enum

1 A `struct` is a collection of named fields. Each field has its own type.

1a **Array fields** — if the type of a field is `array n t`, that field holds the bytes of `n` elements **inside the record** (since the length is in the type, the place is fixed). `t` is a sized number or `bool` — fields whose elements are structs or arrays this processor has not yet built (E-LIT-UNBUILT). The sum of the bytes of the array fields of one struct has the same limit as a frame (E-FRAME-SIZE, §6.2.6 (7c)). `field r. body .` is a slice that sees those bytes, and `set idx field r. body . i. . v. .` writes an element of the record. A list literal given to the field shall have the same element type and length as the field (E-TYPE-FIELD). In a byte layout (`view` · `encode` · `size_of` · the C-side layout) an array field lies **as it is inside the struct bytes**, as in C — its size is `n × ¦t¦` and its alignment that of `t` (`len u8 . body array 4 u8 . tail u16 .` is 8 bytes — `len` from byte 0 · `body` from byte 1 · `tail` from byte 6). The array field of a view is a slice that sees those bytes, and writing the field copies into that place.

2 An `enum` is one of several. Each variant has a name and may carry values.

2a Each variant **is closed with `..`** A newline is not a closer, so variants written one per line without dots run together into one variant and are refused (E-ENUM-DOT). The values a variant carries are written as `<field name> <type>` pairs — if the pairs do not match, it is refused (E-ENUM-FIELD).

2b An enum value is not a number — to use a variant's number as a number, write `cast` (§6.2.16 (5)).

2c **Every variant has a number.** The first variant is 1, and the next variant is the number of the previous variant plus 1. An integer written after a variant name (`ga 11 .`) becomes that variant's number, and the next variant counts on from there. The numbers shall **increase** in the order declared — a number not greater than the previous variant's number is refused (E-ENUM-NUMBER). Numbers are written only in an enum with no variant that carries values (E-ENUM-NUMBER). 0 is not the first variant's number — in an enum whose numbers are not written, 0 is no variant.

Example — numbers written on variants

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 .
```

Output

```
total() = 226
```

Rejected example — the numbers do not increase

Source

```
module ex_enum_number_falls .

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

Diagnostic: `E-ENUM-NUMBER`

2d **The width of an enum.** An enum whose variants carry no values may have one line `layout <width> .` in its block. The width is one of `u8` · `u16` · `u32` · `u64`, and is the number of bytes the enum occupies when it enters a byte layout (§6.2.13) — the bytes hold the variant's **number** (2c). This line is not a variant. An enum without a width has no byte layout. The following are refused (`E-ENUM-LAYOUT`): a width outside the four, a variant number that does not fit the width, an enum with a variant that carries values, two `layout` lines. ☞ Why the width is not chosen automatically: adding one variant must not silently change a layout beyond a wire or a file.

Example — an enum with a width is laid in one byte as its number

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 is the number of no variant
  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 .
```

Output

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

Rejected example — a number does not fit the width

Source

```
module ex_enum_layout_fit .

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

Diagnostic: `E-ENUM-LAYOUT`

3 A struct **cannot have itself as a field.** If it cycles, directly or through other types, translation is refused (`E-STRUCT-CYCLE`). Because its size would not be fixed.

3a If it points to itself through a reference, the size is fixed. But if that reference is one **whose manager cannot be known**, it is refused (`E-STRUCT-REF-UNMANAGED`) — having a size does not mean the lifetime is settled.

3b If the type of a field is exactly `option (owned <that struct>)`, it is not regarded as the cycle of (3). That field is not a value but **one address slot** pointing at another node built in allocator bytes (`none` when empty), so the size is fixed. Such a struct is called a node that holds itself, and the rules for building and giving it back are specified by §8.5 (10)–(13). `owned <itself>` without `option`, and ownership cycles that come back through another struct, are still caught by (3).

Example — struct and enum

Source

```
module ex_shape .

struct point do
  x u32 .
  y u32 .
end .

enum color do
  red .
```

```
  green .
end .
```

4 A struct value is made with `lit`. When made, **every field shall be filled**. `_ <value> .` on the last line puts that value into every field not written (`lit pt do y a. . _ 0 . end .`) — the value is one name or literal (`E-LIT-INDEX`); if a field remains that the value cannot go into, it is refused naming that field (`E-TYPE-FIELD`); and if there is no field to fill, it is refused (`E-LIT-COUNT`).

4a **A struct is a value.** When it is made or copied (`var q point p. .`), the bytes of its array fields and the other struct values in its fields are copied with it — changing the copy does not change the original. Actor instances (things with an identity) and structs with `owned` fields are not copied but point at the same thing.

4b **Writing a field is writing to that record.** `set field r. x . v. .` and writing an element of an array field are accepted only when `r` is a writable place: a `var` or a `mut` parameter. Writing to a record bound with `let` is refused (`E-IMMUTABLE`) — so is taking an array field out as a writable slice (`mut slice`). Writing a field of a parameter received by value writes to that op's **local copy** — the caller's record does not change (§6.5.1 (2a)).

5 A field is read with `field <value> <field name>`. There is no form that attaches a dot and a field name after the value (`E-FIELD-GLUED`) — the dot is already used for module qualification and variant names.

Example — making and reading a struct

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 .
```

Example — copying a struct with an array field

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 .
```

Rejected example — writing the array field of a record bound with let

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 .
```

Diagnostic: `E-IMMUTABLE`

Rejected example — variants not closed with dots

Source

```
module ex_enum_dot .

enum color do
  red
  green
end .
```

Diagnostic: `E-ENUM-DOT`

Note (informative)

Cycles are forbidden because of size. A struct that holds itself would have infinite size. If a tree-like data structure is needed, link it by **numbers (indices)** — then the size is fixed, and bounds checks hold as they are.

6.2.8 Types that hold an answer — result and option

1 `result t e` holds either a successful value (`t`) or an error (`e`).

2 `option t` holds either a value or none.

3 Values of these two types **shall be checked for which side they are before being taken out.**

3a Taking out is a **partial operation** — taking out without checking is not itself refused at translation, and taking out the absent side during execution **traps**. That is, the check is not done for the author by translation; the author does it.

3b The type of the contained value is part of the type. `option usize` and `option u64`, `result u32 e` and `result u64 e` are different types, and putting one in the place of the other is refused at translation (`E-TYPE-NOMINAL` · `E-TYPE-WIDTH` · `E-TYPE-SIGN`). The type of a value taken out (`some_value` · `ok_value`) is the type of the contained value. When `some` · `ok` wrap a literal, it follows the type of the receiving place.

3c A value of `result` · `option` is not an operand of arithmetic or comparison (`eq` · `lt` …) (`E-TYPE-KIND`). Write what is being asked — `is_some` · `is_ok` · `value_or`.

Example — returning failure with result

Source

```
module ex_result .

enum err do
  too_small .
end .

rem If it is less than 2, report that it cannot be halved.
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 .
```

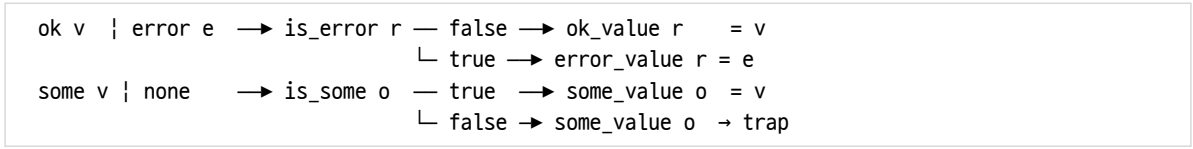
In plain words

There is a custom of using **special values** such as −1 or a null pointer to signal failure. Then *"is this −1 an error or just −1"* cannot be known from the source alone. `result` and `option` move that question into the type, so that the processor asks it instead.

4 The making side wraps values with `ok` (success) `error` (error) `some` (a value is present) `none` (no value).

5 The receiving side first asks which side it is — `is_error` (is it an error) `is_some` (is a value present). Then it takes out — `ok_value` (the success value) `error_value` (the error) `some_value` (the present value).

Diagram — wrap · ask · take out



6 `value_or` gives the value if there is one, otherwise the value given in its place. It reduces the two steps of asking and taking out to one.

7 `try` takes a `result`, takes out the value on success, and on failure **passes that error on upward as it is** (§6.5.7).

8 If a `result` is received and **nobody looks at it**, the processor reports it (`W-RESULT-DISCARD`). The design of returning failure as a value is kept only when the receiving side **looks** at that value. There are two shapes of not looking — calling the op **as a statement** and discarding the value without even a name, and **putting it into a name and never reading it**.

8a A place that **deliberately lets** a failure pass is written `drop <name> ..` Then it is not reported. There are places where there really is nothing to do with a failure — such as the place on an error path that closes resources and leaves. `drop` already means «I end this here», so no new word is put in that place.

In plain words

Why a report and not a refusal? Ignoring a failure is not **always** wrong — what is wrong is ignoring it **silently**. So what this clause requires is not «do not ignore» but «write that you ignore», and once it is

written, a reader reads it as **the author's judgment**. What is not written, nobody knows whether it is judgment or a mistake.

Example — taking out without checking stops during execution

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 Take it out at once without checking — translation passes.
fn raw input k u8 . output u8 .
do
  return some_value mk k. . . .
end .
```

Output

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

In plain words

`raw(2)` gives 20. `raw(7)` takes out a value that is not there, so it **stops** — it does not silently give 0. Translation does not prevent this in advance because on which path a value is present is **what the author knows**, and the processor cannot always know it. Instead, when it is wrong, **it is not silent.**

Example — option — the making side and the receiving side

Source

```
module ex_option .

rem The making side — some if there is a value, none if not.
fn lookup input k u8 . output option u8 .
do
  guard lt k. 3 . else return none . .
  return some mul k. 10 . . .
end .

rem The receiving side ① — ask, then take out.
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 The receiving side ② — if absent, give a value to use instead.
fn use_or input k u8 . output u8 .
do
  return value_or lookup k. . 99 . .
end .
```

Output

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

In plain words

There is one dot in `output option u8 .` — `option` takes **one** type, so it ends at that number of arguments, and the dot closes the `output` clause. The old shape that doubled a dot for every type (`option u8 . .`) is refused (§6.1.6 (2e)).

Example — result — asking and taking out, and passing on with `try`

Source

```
module ex_result_use .

enum io_error do
  too_big .
end .

rem The making side — when it fails is written in the contract.
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 The receiving side ① — ask, then take out.
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 The receiving side ② — try passes the failure on upward as it is.
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 .
```

Output

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

In plain words

Note that `use_try` writes `errors` itself too. To pass a failure on with `try`, it **shall be able to return that error itself**, and it shall write that in its contract. There is no path by which a failure silently disappears.

6.2.9 Named types

1 `type` gives an existing type another name. The two names are **the same type** and can be used interchangeably.

2 `newtype` makes **a different type** with the same representation as an existing type. They cannot be used interchangeably.

2a The shapes are `type <name> <type> .` and `newtype <name> <type> .`. No word is inserted between the name and the type.

Rejected example — a `newtype` does not mix with its base type — tests (2)

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 .
```

Diagnostic: `E-TYPE-NOMINAL`

2b The type of `type` and `newtype` cannot itself be a **row type** — `slice` · `array` · `matrix`. The same holds with `mut` in front. A row type is written as it is where it is used. A row type **inside** another type can be given a name (`type found option slice u8 .`). If a type with its own name is needed, put it as a struct field (§6.2.7).

Table 21 — Diagnostics for naming a row type

Violation	Diagnostic
(2b) — the whole type is a `slice`	`E-TYPE-ROWNAME`
(2b) — the whole type is an `array`	`E-TYPE-ARRAY`
(2b) — the whole type is a `matrix`	`E-MATRIX-USE`

Rejected example — another name given to a slice — tests (2b)

Source

```
module ex_rowname .

type bytes slice u8 .

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

Diagnostic: `E-TYPE-ROWNAME`

In plain words

A row type shows its element type and its length · dimensions in its words. Hidden behind another name, a reader has to go to the declaration to know them — and there become two ways of writing the same type. A struct field is different: a different name makes a different type, so that name does not hide but **separates**.

Note (informative)

The two are separated to prevent mistakes. If user numbers and order numbers are both `u64`, the processor does not notice when they are swapped. Separated with `newtype`, the processor catches that mistake.

3 When a `newtype` is used as **a name that separates storage** (*brand*), one brand points at **one** storage. Opening storage twice with the same brand is refused (`E-BRAND-REUSED`) — then one name would point at two, and handles from different storage **would look like the same type.** That is exactly the confusion the brand was meant to prevent. If they must be separated, declare one more `newtype`.

6.2.10 Reading inside a value — field

1 `field` is the form that reads what is inside a value. Its shape is `field <value> <step> …`, with **one or more** steps. With several steps it goes down one level at a time from left to right — `field o. inner deep .` means `inner` of `o`, and `deep` inside that.

2 A step is one of two things. A **name** is a field of a struct, and an **integer** is a position in a row (array · slice). The two may be mixed.

3 `field` is also a *place* — it can come on the left of `set`. The reading place and the writing place have **the same spelling**.

4 If a field not in the declaration is written, the processor refuses the program. Positions in rows follow the bounds rules of §6.2.6.

> **Example — reading and writing a multi-level field**

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 .
```

> **Note (informative)**

The notation with attached dots (`o.i.a`) **does not exist**. If operations on values are made to look like name paths, the meaning is decided only once the reader knows whether `o` is a value or a module — and that is the work of name resolution, not of grammar. Written as a form, it is decided **the moment it is read**.

6.2.11 Values with lanes — vec

1 `vec <lane count> <element type>` holds several values of the same type **as one value** (the same order as §6.2.6 (1a) — the lane count comes first; the old order `vec <element type> <lane count>` is `E-TYPE-ORDER`). `scalable` can also stand in the lane-count position (`vec scalable u8`). The lane count shall be a value fixed at translation time (§6.8).

2 Two `vec`s with different lane counts are different types and do not mix.

3 Arithmetic on a `vec` happens **per lane**, separately. Values do not cross between lanes.

3a The per-lane arithmetic is `add` · `sub` · `mul` · `div` · `mod` · `neg` · `abs` · `min` · `max`. Overflow of integer lanes **wraps** at the lane's width (unlike scalars — the quotient of the smallest value of a signed lane divided by −1, and its `neg` · `abs`, are that value itself). If an integer lane's `div` · `mod` has a lane whose divisor is 0, it stops as for scalars. The quotient is truncated towards 0, `mod` follows the sign of the divisor (§6.3), and `min` · `max` of signed lanes compare by sign (2026-11-02). Floating-point lanes have no `mod` — as for scalars, translation is refused (§6.3.3 (2a) · `E-TYPE-KIND`).

> **Note (informative)**
>
> If the machine can compute several lanes at once it does so; if not, it computes them one by one. **Either way the answer is the same** — the lane count is not a performance hint but part of the type.

> **Rejected example — the lane count written after the element type**
>
> **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 .
> ```
>
> Diagnostic: `E-TYPE-ORDER`

6.2.12 Sets made of bits — `bitset`

1 `bitset <bit count>` holds whether each value from 0 up to, but not including, that count is present. The bit count shall be a value fixed at translation time.

2 Inserting or asking about a value outside the range does not conform.

3 A `bitset` is **a set**, not the bits of a word. Handling the bits of a word is bit operations (§6.3), and the two are different things.

4 Inserting an element is `bitset_insert <set> <value> .`, removing one is `bitset_remove <set> <value> .` — both change the set in place. `add` means only the addition of numbers, so using it on a set does not conform (`E-BITSET-ADD`).

4a The number of elements in a set is `bitset_count <set>`, and its type is `usize`. The number of elements of a stack is `stack_count <stack>`. Using one in place of the other, or using `len` (the length of a piece) on a set · stack, is refused at translation (`E-TYPE-ARG`).

> **Example — inserting into and removing from a set**
>
> **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 .
> ```
>
> **Output**

```
members() = 1
```

6.2.13 An eye laid over bytes — views

1 A byte slice can be read, **without copying**, as if it were an array of another type. This is called a view.

2 When a view is obtained, the alignment and length contract is checked. If the contract is broken, the program stops.

3 A view **does not copy** bytes. Writing through a view changes the original bytes.

> **Caution — A view counts its length in elements**
>
> The length of a byte slice is the number of bytes, but seen as an array of four-byte elements its length is **the number of elements**. Seeing the same storage with two eyes, `len` answers differently — because they are counting different things.

4 The opposite of a view is `encode`. `encode <shape> <value>` **writes** the value **into bytes** in the layout the shape specifies. If laying over and reading is `view`, writing down is `encode`.

5 **A view laid over a struct is a window.** `view <struct> <bytes>` gives a **window** that lays the layout of that struct over a byte slice. Reading a field of the window reads the bytes at that place, and writing a field writes the bytes at that place — it is not a copy. The type of a window is `view T` (a read window) or `mut view T` (a write window), different from the type `T` of a struct value. `try_view` does the same but gives `none` if the length · alignment do not fit or a field does not fit (9a) — its type is `option view T · option mut view T`, and `some_value` is the window. The conditions on length and alignment are specified by §8.7.2.

6 **Windows and values do not mix.** Putting a window in the place of a struct name `T · option T` — binding · returning · putting a whole struct value into a window name — is refused at translation (`E-VIEW-COPY`). A **copy** that lives apart from the bytes is made with `cast T <window> .` (§6.2.16 — an explicit conversion is `cast`). Binding a window to another window name sees the same bytes. A window can also be passed to an input that takes a value (`input p T .`) — the receiver only reads.

Table 22 — Windows and values

Written	What it is
`let h view pt view pt b. .`	a read window over `b`. Reading a field reads `b`
`let h mut view pt view pt b. .`	a write window over `b`. `set field h. y . 9 .` changes `b` — `b` shall be writable
`let k mut view pt h. .`	another write window that sees the same bytes
`var k pt cast pt h. . .`	a struct value that copied the fields of `h`. Writing to `k` leaves `b` as it is
`put h. .` (the input of `put` is `mut pt`)	the bytes of `b`. If the receiver writes a field, `b` changes — it shall be a write window
`output view pt · output mut view pt`	returns a window that sees the bytes of an input

7 **Whether something can be changed is written in its type** (§8.8 (2)). A write window comes only from writable bytes — binding a write window from a shared input, a slice bound without `mut`, a string literal, a read window, or the result of an op that gives a read window is refused (`E-TYPE-MUT`). Writing a field through a read window or doing `write_volatile` is `E-TYPE-MUT`, passing a read window to a `mut` input is `E-TYPE-ARGMUT`, and returning a window over read-only bytes when the output is `mut view T` is `E-TYPE-RETMUT`. If an `fn` writes through a window over bytes it received as `mut` —

whether laid directly or returned by another op — that is visible to the caller, so it is not pure (`E-EFFECT-PURITY`, §7.3 (6)).

8 A struct a view can be laid over is a struct **whose byte layout is fixed**. Each field shall be a number of fixed width (`u8`…`u64` · `i8`…`i64` · `f32` · `f64`), a `bool`, an enum with a written width (§6.2.7 (2d)), a fixed array of those, or a struct made of such fields. Enums without a written width · slices · `option` · owned-handle fields have no fixed bytes, so neither a view nor `encode` can be used on such a struct (`E-IR-UNDEF`).

9 Every field of a struct read with `view` and `view_array` shall be `plain` (§8.7.1 (2)). A `bool` field that receives a byte other than 0 · 1 becomes a value that is neither true nor false, and an enum field that receives a number belonging to no variant becomes a value that is no variant, so if there is even one such field (including in inner structs) it is refused (`E-TYPE-BITCAST`) — for the same reason that `bit_cast bool` is refused. `view_array bool` is the same. `encode`, which writes values down into bytes, has no such restriction.

9a `try_view` **measures** such fields — if the byte of a `bool` field is not 0 · 1, or the number of an enum field is not the number of any variant of that enum, it gives `none`. It measures the fields of inner structs too. Reading an enum field of a value obtained that way gives the variant of that number, and writing an enum field puts that variant's number into the bytes. If, after measuring, the underlying bytes change by another path so that they are no variant's number, execution stops the moment that field is read.

Example — writing a field of a view changes the bytes, and binding from name to name copies

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 becomes [1,9]
  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 a copy that is a struct value
  set field k. y . 7 .
  return idx b. 1 . .      rem b is unchanged
end .
```

Output

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

Rejected example — binding a write window from bytes received read-only

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 writing a field would change the caller's b
  set field h. y . 9 .
  return 0 .
end .
```

Diagnostic: `E-TYPE-MUT`

Rejected example — binding a window in the place of a struct value

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 a window or a copy? — `view pt` or `cast pt`
  return field h. y . .
end .
```

Diagnostic: `E-VIEW-COPY`

Rejected example — a `bool` field can receive a byte other than 0 · 1

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 .
```

Diagnostic: `E-TYPE-BITCAST`

6.2.14 Parameters with a narrowed range of values — `range`

1 If `range <low> <high>` is written in the type position of a parameter, that parameter accepts only values between the two ends, both included.

2 If the caller cannot prove that the value is within that range, translation is refused. A value that came from **outside** the program (an argument of the entry point and the like) has no caller to prove it, so it is checked **at the boundary**, and the program stops if it is out of range.

2a If the caller's type is **wider** than the range, that alone is refused (`E-TYPE-WIDTH`). A value of the same width, which is not caught by that check, has its range checked **at the op's entry** and stops if it is out of range (`E-VM-CONTRACT`) — the same check as writing `requires ge <name> <low> .` and `requires le <name> <high> .`.

3 A declared range is used with the same standing as a contract of §6.4 — the checker may take it as a fact and remove the checks that follow.

> **In plain words**
>
> (3) **makes** (2a) **necessary**. If the checker takes the range as a fact and removes bounds checks, while nobody keeps that fact, values outside the range pass through where the checks were removed. Until 2026-09-16 a `range` parameter of the same width was checked at no layer (defect note 10) — **the only thing to trust is what is checked.**

> **Rejected example — the caller's type is wider than the range**
>
> **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` is wider than 0 … 100 — it was not shown to be within the
> range
> end .
> ```
>
> Diagnostic: `E-TYPE-WIDTH`

> **Note (informative)**
>
> The same thing can be written with `requires` too. The difference is **where it is written**. Written in the type position, the range becomes the **shape** of the parameter, and the caller knows it from the signature alone. Written as a contract clause, one has to read the contract to know it.

6.2.15 Nailing down the byte layout — `layout` and `align`

1 A layout can be written inside a struct declaration. If none is written, fields are placed **in declaration order**, each aligned to its type's alignment (with as much padding between as alignment requires). The processor does not change the order.

2 `layout packed .` **puts no padding** between fields — the fields are packed together in declaration order.

2a The layout words are a closed two — `packed` and `native`. Anything else is refused (`E-TYPE-LAYOUT`).

3 `align <n> .` nails down the alignment of the place where that struct begins. n shall be a power of 2. The alignment shall be **a power of 2** (`E-TYPE-ALIGN`) — no machine can keep any other number.

> **marker** (표지)
>
> A word attached after a field that fixes **how that field is placed and how it can be reached**. It is not a type, so it does not change the size of the value; it fixes only the byte order and the way of reaching.

4 A *marker* can be attached after a field. The markers are these five and no more; writing any other word is refused at translation (`E-FIELD-MARK`).

Table 23 — Field markers — a closed set of five

Marker	What it fixes	Meaning
`big`	byte order	placed **big-endian**
`little`	byte order	placed **little-endian**
`rw`	way of reaching	can be read and written
`ro`	way of reaching	**read only** — writing is refused at translation
`wo`	way of reaching	**write only** — reading is refused at translation

4a If no byte order is written, the machine's order is followed (§5.6).

4b The way of reaching is enforced only in **structs that mirror device registers** (§7.6). A violation is refused **at translation**, not at execution (`E-MMIO-PERM`).

> **In plain words**
>
> Why is reading `wo` an error? Reading a write-only register yields garbage, or the act of reading itself moves the device. That is, that read is **not useless but wrong**. Once the device has said what it accepts, the type says it back.

> **Example — a header going out on the wire — no padding, big-endian**
>
> **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 .
> ```

> **Note (informative)**
>
> **Nailing the layout down and not nailing it down buy different things.** Not nailed down, the processor can choose to fit the machine, and it is fast. Nailed down, those bytes have **the same meaning outside too** — what goes out on a wire or is read by a device needs that. So nail it down only where needed.

6.2.16 Explicit conversion — `cast`

1 **Reading a value as another type happens only where the source says so.** The only conversion the processor does by itself is widening (§6.2.5 (2)); every other conversion is written by the author with a word. The centre of those words is `cast <type> <value>` — **`cast` is the word that takes charge of explicit conversion in this language.** `widen` is used only where **the value does not change**, and conversions that may change the value are written with `cast` — places that change the sign, cross kinds, or narrow.

1a There are three words that write conversions, each with its own place.

Table 24 — The three words that write conversions

Word	Its place	If the value does not fit
`widen`	widening in which the value fits **as it is** (the table of ⟦§6.2.5⟧ (4)). The processor also does it by itself — writing it makes it visible	does not happen
`narrow`	reduces the width within the same kind (integer → narrower integer)	traps
`cast`	**every other conversion** — changing the sign · between integers and floating point · between `usize` · `idiff`, whose width depends on the implementation, and other integers · truth value → integer · enum ↔ integer	traps

1a-1 A `narrow` that crosses kinds (integer ↔ floating point) is refused at translation (`E-TYPE-KIND`), and so is one whose target is wider than the value (`E-TYPE-WIDTH` — the word for that is `widen`).
1a-2 `narrow f32` rounds an `f64` value to `f32` **once** (the same rounding as (3-1)). If a finite value rounds beyond the largest finite `f32` — the value does not fit `f32` — execution stops. Infinities and NaN pass as they are. To take an overflowing value as the infinity of the same sign, use `cast f32`.
1b **A conversion that could be read either way does not happen by itself; it goes through `cast`.** That is a conversion where the same bits could be read as two numbers, or that widens or narrows depending on the target machine, or that changes the kind of the value. This rule also applies to types and conversions that enter the language in the future — if a new conversion is ambiguous its spelling is `cast`, and nothing is decided in the direction of adding implicit conversions.

> **In plain words**
>
> When conversions are written in the source, a reader **can find every place where a value may change with their own eyes.** Having found and read the `cast`s, they have seen every place in that program where a type changes. If there were even one place where the processor changed it secretly, that promise would be broken. So this language has no implicit conversions for convenience. Writing them takes a little effort, but when reading there is no guessing «what changed here».

2 `cast` **traps if the value does not fit the target type.** It is not truncated silently. That is, `cast` is not a mark that says «anything may happen» but a mark that says *"I know this is a place where the value may change"*.
3 Converting a floating-point number to an integer **truncates towards 0** — `7.9` becomes `7` and `-2.9` becomes `-2`. Only the fractional part is discarded; if the integer part does not fit the target type, it traps as in (2).

> **Caution —**
>
> Discarding the fractional part **does not trap.** So `cast` does not violate *"there is no implicit conversion that loses a value"* of §6.2.5 (1) — because that conversion **is not implicit.** The author wrote `cast`, and that word itself is the mark that says «the value may change here».

3-1 **A `cast` to a floating-point type** — whether from an integer or from another floating-point type — gives **the nearest value** in the target type, and when two are equally near, the one whose last digit is even. Rounding happens **once** — converting an integer to `f32` does not go through `f64` and round twice. A value that rounds beyond the largest finite value of the target type becomes the infinity of the same sign, and NaN is NaN. This conversion does not trap as in (2) — a floating-point type has an infinity for an overflowing value to go to. From `f32` to `f64` the value is unchanged. `cast f32 <f64 1e300>` is +∞, and `cast f32 <u64 16777217>` is `16777216`.

3a **Between `usize` · `idiff` and other integer types** the only crossing is `cast` (§6.2 (3-3)). This conversion is both a widening and a narrowing depending on the implementation's width, so `cast` is its spelling — if the value fits it is given as it is; if not, it traps as in (2). On an implementation where the width is the same or grows, this `cast` leaves no check.

4 **From a truth value to an integer** goes by `cast` — `cast <integer type> <truth value>` is `1` for true and `0` for false. That value fits every integer type, so the trap of (2) does not happen. A floating-point type is not a target of this conversion (`E-TYPE-KIND`).

4a There is no `cast` **from an integer to a truth value** — `bool` cannot be the target of `cast` (`E-TYPE-KIND`). Write what is being asked: `ne n. 0 .` or `eq n. 0 .`. Which values of the number are true is decided by that question.

4b Truth values are not in the table of `widen` (§6.2.5 (4)) — `widen <integer type> <truth value>` is refused (`E-WIDEN-KIND`). The spelling that turns a truth value into a number is `cast` alone.

Example — a truth value as a number

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 .
```

Output

```
big(5) = 1 · big(1) = 0 · count_big(5, 7) = 2
```

Rejected example — there is no cast that turns a number into a truth value

Source

```
module ex_cast_to_bool .
fn nonzero input n u64 . output bool . do
  return cast bool n. . .     rem write what is being asked — `ne n. 0 .`
end .
```

Diagnostic: `E-TYPE-KIND`

5 **An enum value is not a number.** Putting an enum value in a place that takes a number — a name · field · argument · return declared as a number, arithmetic, comparison with numbers, indices, bit operations, widening · narrowing — is refused at translation (`E-TYPE-KIND` · `E-TYPE-LET` · `E-TYPE-RETURN` · `E-TYPE-ARG` · `E-WIDEN-KIND`), and putting a number in a place that takes an enum is refused too, literals included. If the variant's number (§6.2.7 (2c)) is needed, write `cast <integer type> <enum value>` — if the number does not fit that type, it traps as in (2). Two values of the same enum can be compared with `eq` · `ne`, and with `lt` · `le` · `gt` · `ge` — the magnitude is the order of the numbers, and the numbers increase in declaration order, so it is the same as the order of declaration. Values of different enums are not compared (`E-TYPE-KIND`).

5a **From a number to an enum** goes by `cast <enum> <integer>` — it gives the variant with that number. If the number belongs to no variant, it **traps** (`E-VM-CAST`). This is the place where a number read from a file or a wire is turned back into an enum. The source shall be an integer (`E-TYPE-KIND`).

5b An enum with even one variant that carries values (§6.2.19) can be neither the source nor the target of `cast` (`E-TYPE-KIND`). One number cannot say everything about such a value — which variant it is is asked with `isa` · `match`, and the value is built with `<enum>.<variant>` ….

Example — a variant's number is read with cast

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 .
```

Output

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

Rejected example — giving an enum value as a number

Source

```
module ex_enum_as_number .

enum color do
  red .
  green .
end .

fn code input c color . output u8 . do
  return c. .     rem a variant's number is read with `cast u8 c. .`
end .
```

Diagnostic: `E-TYPE-RETURN`

Example — turning a number read back into an enum and filtering by range

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 = weapon · 2 = tool — values of the same enum are compared by magnitude
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 .
```

Output

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

Rejected example — comparing values of different enums

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 .
```

Diagnostic: `E-TYPE-KIND`

Rejected example — doing arithmetic with an enum

Source

```
module ex_enum_arith .

enum level do
```

```
  low .
  high .
end .

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

Diagnostic: `E-TYPE-KIND`

6.2.17 What holds characters — a string is not a type

1 This language **has no string type.** What holds characters is a piece of bytes (`slice u8`), and that is all.

2 So **the length is the truth.** A mark that signals the end of a string (a null byte) is not part of the value, and scanning the bytes to count the length does not happen.

> **In plain words**
>
> A null-terminated string is a design that **hides the length inside the value**. So to know the length one has to scan, and if the mark is missing while scanning, what lies beyond is read — in practice the most common memory accident came from here. A piece **carries its length outside**, so that accident has no place to stand.

3 The null-terminated shape is needed only where C is met (§6.9.2). It is **a shape built at the boundary**, not a type this language has.

4 Handling characters — cutting · searching · joining · Unicode checks — is **the work of libraries** (§9.5). The processor knows only pieces and bytes.

6.2.18 Type names without meaning

1 Some names **are accepted but have no meaning.** If such a name is written in a signature, the processor warns (`W-NOT-YET`). Those names are gathered in annex A.9 (2) — names such as `byte` · `char` · `str` · `string` · `bytes_view` · `dyn` · `atomic`.

2 Having no meaning means that **the processor does not know the shape of the values that name holds**. Neither the size, the layout, nor what can be done with the value is fixed, so checks that rely on that name **do not stand**.

> **Caution —**
>
> This is why this place **shall speak**. Translation passes and answers come out, so the checks that did not stand show nowhere. **What silently dropped out, nobody finds** — so the processor says so.

3 There are two ways to give a name meaning: **write the underlying type directly**, or declare that name (`type str option u64 .`). Once declared, it has a meaning, so there is no warning.

> **Example — a name without meaning is warned about but not refused**
>
> **Source**
>
> ```
> module ex_name_only .
>
> rem `str` is not built in — without a meaning it gets W-NOT-YET. Here it is given a meaning by
> declaring it.
> type str option u64 .
>
> fn f input s str . output u8 . do return 1 . end .
> ```

Output

```
f() = 1
```

6.2.19 Variants that carry values — building and taking apart

1 The variants of an `enum` can carry values. When writing a variant, the names and types of what it carries are written with it.

2 The shape that builds such a value is `<enum>.<variant>`, and it gives **all** of what the variant was declared to carry. If something is missing it is refused (`E-ENUM-ARITY`) — a half-built value is not made silently.

3 There are two ways to handle an enum that carries values.

Table 25 — The two ops that handle variants carrying values

op	Shape	What it does
`isa`	`isa <value> <variant>`	asks whether it is that variant now
`payload`	`payload <value> <variant> <name>`	reads what that variant carries

4 Naming a variant that does not exist is refused (`E-ENUM-NOVARIANT`), and trying to read a name that variant does not carry is refused (`E-ENUM-NOFIELD`). Both are decided **at translation** — the variant and the name are written in the source, so there is no reason to wait for execution.

5 In an enum with several variants, `payload` can be used **only after it has been narrowed to that variant** (`E-ENUM-UNCHECKED`). What narrows is `isa`, usually written as `guard isa <value> <variant> else ….`

5a Using `payload` at a place already narrowed to **another variant** is refused too (`E-ENUM-VARIANT`) — at that place it has already been proven to be another variant, so that `payload` **reads the wrong thing.**

5b The type of what a variant carries is **written as one word** (`E-ENUM-PAYLOAD`). A variant line has no mark that separates fields, so for types written as several words (`slice u8 · option t · owned t · ref t`) where they break would not be fixed. This is a restriction of spelling, not of size — the size of the carried value is not asked about. A multi-word type is written by naming a struct that has it as a field and using that name.

In plain words

What (5) does is prevent «reading what is not there». Reading without narrowing is right only when the value is that variant, and when it is not it reads **unrelated bytes** — that is not an answer being wrong but an answer that is **right by chance**, the shape this language dislikes most.

Example — a multi-word type is carried through a named struct

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 .
```

Output

```
f([1,2,3]) = 3 · g() = 10
```

6 If an enum carries itself as a value, its size is not fixed, so it is refused (`E-ENUM-INFINITE`). A recursive structure is built by **putting something in between**.

> **In plain words**
>
> A variant that carries values is not a new construct. There is just one field that indicates the variant and an aggregate holding what the variant carries; building is building an aggregate, and reading is reading a field. So this feature **added not a single machine instruction** — it lowers to what already existed, so it ran on both back ends by itself.

6.2.20 Seeing scattered pieces as one — `segments`

1 Several pieces that are not contiguous can be seen with **one eye**. It consists of two things: the bytes lying behind, and a descriptor that says where in them and how much.

Table 26 — The ops that handle scattered pieces

op	Shape	What it does
`view_segments`	`view_segments <bytes> <descriptor>`	binds the two into one eye
`segview.piece_count`	`segview.piece_count <eye>`	counts how many pieces there are (standard module `segview`)
`segview.piece_at`	`segview.piece_at <eye> <number>`	gives the piece with that number as a piece (slice) (standard module `segview`)

1a An eye is an aggregate holding the fields `back` (the bytes behind) and `descs` (the descriptor). The number of pieces and the piece of a number are written by reading and cutting those fields, so they are not operations of the language but ops of the standard module `segview` (§9.3).

2 A descriptor is a piece holding pairs of «from where» and «how much». So these three **add neither a new type nor a new machine instruction** — they lower to building an aggregate, reading fields and cutting.

> **In plain words**
>
> Without a way to handle scattered pieces, the user has no choice but to **gather them into one place**. Gathering is copying, and copying where it need not be done is the cost this language sets out to remove. Given an eye, the copying disappears.

6.2.21 Fetching ahead — `prefetch`

1 `prefetch <piece> <number>` tells the machine that the bytes at that place **will be used soon**.

2 This is **only a hint**. It leaves no value and does not change the meaning of the program, and if the machine ignores it, the answer is the same.

> **Caution —**
>
> A hint is a hint only if **being wrong does not make the program wrong**. Something that changes the meaning is not a hint but a command, and such things do not come in this place.

6.2.22 What it means for types to match

1 In every place where a value is put, the type of the value shall be **the same** as the type that place states.

Table 27 — Places where types do not match

Diagnostic	Where
`E-TYPE-LET`	binding with `let` — the value given differs from the type written
`E-TYPE-VAR`	binding with `var` — the same mismatch, at a different place
`E-TYPE-SET`	rewriting — the new value differs from the type of that name
`E-TYPE-RETURN`	returning — what is returned differs from what the op states
`E-TYPE-ARG`	passing — an argument differs from the parameter
`E-TYPE-FIELD`	**making** — a field that does not exist is given, a field's value differs from its type, or a field is missing
`E-TYPE-TRY`	`try` — a value with nothing to strip off

2 Aggregates and enums are declared with `struct <name> do … end .` · `enum <name> do … end .`. Trying to declare them in another shape is refused (`E-TYPE-DECL`).

3 Even with the same field layout, **different names are different aggregates** (`E-TYPE-STRUCT`, §8.9).

> **In plain words**
>
> The seven in this table are all different places of the same sentence — the source wrote *"the type of what comes here is this"*, and the processor compares what is written with what actually comes. There is a separate diagnostic for each place because **where it mismatched** is the fact the person fixing it needs most.

6.2.23 The shape of writing fields

1 A field of an aggregate is **a name followed by a type** (`x u64 .`). Anything not in that shape is refused (`E-FIELD-FORM`).

2 A field is read with `field <value> <name> ….` The old spelling that reads with an attached dot does not exist in this language (`E-FIELD-GLUED`).

3 A `vec` and a mask are **values themselves**, and pointing or borrowing markers cannot be attached to them (`E-VEC-QUAL`). Their lanes are positions inside a word, not positions in memory.

> **Rejected example — a field is a name followed by a type**
>
> **Source**
>
> ```
> module ex_field_form .
>
> struct p do
>   x .              rem there is no type
> end .
>
> fn f output u8 . do return 1 . end .
> ```
>
> Diagnostic: `E-FIELD-FORM`

6.2.24 Integers of arbitrary width — `bits`

1 `type <name> bits <number> .` makes an integer type with that many bits. The width is **from 1 to 64**.

2 The width need not be a power of 2 — `bits 3` and `bits 10` both stand.

> **Caution —**
>
> If the width does not match the machine word, **speed is not guaranteed.** What this language guarantees is **correctness** — `bits 10` gives the answer computed in ten bits, but it does not say that this is faster than a sixteen-bit computation. To avoid making places whose cost is invisible, what is not guaranteed **shall be written down as not guaranteed**.

6.2.25 Decided but not built: type words — `c8`…`c64` · `b8`…`b128` · `list` · an overlapping sum · a child region

1 The words of this clause are, like the four of §6.9.4, **decided in meaning but not yet built by the implementation**. The three whose spelling is fixed ((2) to (4) below) are refused at translation when written in a type position (`E-TYPE-UNBUILT`) — they are not taken as a name without meaning or turned into another type. None of them is a reserved word: if the author declares a type of that name (`type c8 u8 .`), that declaration stands, and so do a local or an op of the same name.

2 `c8` · `c16` · `c32` · `c64` are **code units of text** — one unit of some encoding, made of that many bits. They are not numbers, so they are not computed with. They go to and from integers only through `cast` (§6.2.16).

3 `b8` · `b16` · `b32` · `b64` · `b128` are **opaque blocks of bytes** — that many bytes, aligned to their size. They have no arithmetic and no comparison; they are only moved and copied. To read one as a number or a structure, go through a view (§6.2.13).

4 `list` is **a list whose elements may differ in type**. Each element carries a mark of its type and the place of its value, and the list is reached only through a handle (it is not a fixed type laid out as a value). An op that uses its elements chooses the branch by the mark.

5 **An overlapping sum** — a type that places one of its variants in the same place; its size is that of the largest variant. The spelling is decided when it is built: the original `union` collides with the rule that the language leaves short words to the author (§6.3.3 (1e) — set union too is `bitset_union`). So this one is not refused as a word.

6 **A child region** — a region carved out of another, which ends when its parent ends (not when a lexical block ends). Carving does not obtain new memory, so it is not the `alloc` effect (§8.13). The original spelling (`child_of`) has no place in the present grammar, so the spelling is decided when it is built.

7 **Deferral.** These words are built when a use appears. If other ways fill that use and they are **shown unnecessary, they are dropped.** What carries that use today: code units — integers (`u8` · `u16` · `u32`) and the `utf8` · `utf16` library; byte blocks — `slice u8` · `array N u8` and views; `list` — an enum whose variants carry values (§6.2.7); the overlapping sum — an enum whose variants carry values; the child region — allocators (§8.13). Whether to build or drop them is the owner's decision. Until then this clause stays and the implementation keeps recognizing the words of (2) to (4).

> **Rejected example — a type word not yet built**
>
> **Source**
>
> ```
> module ex_unbuilt_type .
>
> fn f input x c8 . output u64 . do
>   return 0 .
> end .
> ```
>
> Diagnostic: `E-TYPE-UNBUILT`

> **Note (informative)**
>
> The sources are RFC-0002 §T3 · §S · §LIST (code units · byte blocks · the overlapping sum · the heterogeneous list) and RFC-0004 (the child region). Both RFCs were adopted and then archived, and these words were not carried into the canon (sweep of 2026-11-11, F5). The owner decided, as for §6.9.4, to «put them in formally, state that they are dropped if they later prove unnecessary, and keep tracking them» (2026-11-12). RFC-0002 itself left as an open question «whether a place that really needs `union` appears (today an enum carrying values covers it)».

6.3 Expressions and operations

1 An *expression* is something that produces a value. A literal is an expression, a name is an expression, and the application of an operation is an expression.

6.3.1 Prefix notation is the default

1 Expressions are written in *prefix notation* — the name of the operation comes first and the arguments follow.

2 Prefix notation **has no precedence rules.** What is computed first is said entirely by the dots and the order — a form is opened by a name and closed by its own dot (§6.1.5), and the inner form closes first.

2a Nesting calls needs no parentheses — in `add 1 mul a. b. . .` the first dot closes `mul` and the second closes `add`. The end of a form is decided by **that form's dot**. The number of arguments an operation takes does not decide the end — whether the number of arguments is right is looked at separately after the form has closed (`E-IR-ARITY` if not).

2b Parentheses are **decoration that makes a grouping stand out** — `add 1 (mul a. b. .) .` is the same expression as that of (2a). A form inside parentheses closes with its own dot too, and the parentheses do not change the meaning.

2c **The order in which operands are computed is a free choice** (§4.3). The processor may compute the operands of an operation in any order. Core ops · builtin ops and ops the author declares are the same. But the following are fixed.

- The operation is done **after all** its operands have been computed.

- Each operand is computed **only once**.
- **Operands that produce effects are computed in the order written, from the left.** Here an effect means an effect other than `panic` (§7) —

being able to stop does not by itself bind the order.
- `and` and `or` are the exception — they compute the first operand first, and if that alone decides the result, they do not compute

the second operand (§6.3.3 (4)).

Free choice — FC-0001 — the order in which operands are computed

The processor may compute, in any order, those operands of an operation **that produce no effect**. That order is the only range of choice — computing all operands before the operation, computing each once, and computing effect-producing operands in the order written are fixed. In most programs the answer is the same whatever the order. There is one time the difference shows — when two operands stop for different reasons, **which stop occurs**. If that difference matters, put what is to be computed first into a name with `let` — statements are executed in the order written.

Example — prefix notation

Source

```
let a u32 add 1 2 . .
let b u32 add 1 mul 2 3 . . .
let c bool lt a. b. . .
```

In plain words

Someone seeing `1 + 2 * 3` for the first time has to **memorise** whether `*` or `+` comes first. `add 1 mul 2 3 . .` has nothing to memorise — the dots say it as it is. That is why this language takes prefix as the default.

6.3.2 Infix notation — the `expr` island

1 Long arithmetic is hard to read, so at a place that begins with `expr` *infix notation* can be used. This place is called an *island*.

2 The notation inside an island turns into **exactly the same meaning as prefix**. An island is only a projection of notation and has no execution cost.

3 Precedence exists **only inside islands**. The following table is all of it.

Table 28 — Precedence inside an `expr` island (strongest at the top)

Level	Operation	Associativity	Prefix with the same meaning
5	`( )` grouping	—	only a grouping mark, not a value
4	`* /`	left	`mul` · `div`
3	`+ -`	left	`add` · `sub`
2	`eq ne lt le gt ge`	**cannot be chained**	the prefix operation of the same name
1b	`and`	left	`and` (short-circuit)
1a	`or`	left	`or` (short-circuit)

4 `and` binds more strongly than `or`. That is, `expr a or b and c` is the same as `or a. and b. c. . .`.

5 Comparisons **cannot be chained.** `expr a lt b lt c` is refused.

6 The table above is **all** of the island. This set is closed and does not grow. Operations not in the table — including bit operations (§6.3.5) · the remainder `mod` · the minimum `min` and maximum `max` — are **written in prefix** even inside an island.

7 There are no unary operations in an island either. To flip a sign, write `expr 0 - a`, or wrap a prefix form in parentheses (`expr (not b. .) or c. .`).

Caution — The island is small not because it lacks something

The only value of infix notation is **readability**, and that holds only when **nobody has to look up** the precedence. The twelve in the table are the same as the arithmetic learned at school, and everyone reads them the same way — nobody disputes that multiplication binds more strongly than addition. Bit operations are exactly the opposite. In C, `a & b == c` does not group as `(a & b) == c` but as `a & (b == c)`. When `a · b · c` are all 6 the former is true and the latter false — the answers diverge. This is a trap that catches even people who have written C for a long time, and many languages fix this precedence differently from one another. So putting bit operations in the island **would not make it easier to read but would increase what has to be looked up.** Prefix notation has no precedence at all (§6.3.1), so the parentheses say the answer — it gets longer but **cannot be wrong.** This language chooses not the shorter side but the side that cannot be wrong (§1.3).

In plain words

A notation projection exists to make things easier to read, not to give every operation a **second grammar**. If there are two ways of writing the same meaning, a reader has to know both — that cost is greater than what is gained by shortening.

Example — infix notation

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 .
```

Rejected example — comparisons cannot be chained

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 two comparisons cannot be chained
end .
```

Diagnostic: `E-EXPR-CHAIN`

Caution — Why comparisons cannot be chained

In mathematics `a < b < c` means «a is less than b, and b is less than c». But in many languages that notation is read with **a different meaning** (first `a < b` is computed to get a truth value, and that is compared again with c). If the same characters mean different things to people and to machines, that is a place that silently goes wrong. So this language refuses it outright.

6.3.3 Core ops and builtin ops

1 The operations whose meaning the language fixes are of two kinds. A *core op* is called anywhere by its name alone and shares **one namespace** with the names the author makes. A *builtin op* is called only after `call_builtin` and has **its own namespace** ((1c) · (1f)). For both kinds the list of names is fixed, and the whole of it is in annex D. Annex D also states, for each operation, **what it takes and what it gives, when translation is refused and when execution stops**.

1a Not all the names of core ops have the same rule. Most cannot be used as local names (§6.1.3), but some are operations **only within a context**, and outside it they can be the author's names. Which is which is stated by annex D.

1b What is listed below **shows the basic ones by kind**; it is not all of them. The full list and the meaning of each operation are in annex D.

1c A builtin op stands only after the head word `call_builtin`: `call_builtin <name> <operand>*`. Its names form **a closed set**, and that is the only place it is called. Which names are such is stated by annex D. Those that reach the outside world — files · the network · processes (`file_*` · `dir_*` · `path_remove` · `path_rename` · `link_type` · `net_*` · `proc_*`), standard input and output (`write_out` · `read_in`), arguments and the environment (`arg` · `env_get`), clocks (`time_now` · `time_local` · `time_sleep`), terminals (`tty_raw` · `tty_read` · `tty_size`), random numbers (`random_bytes`), the reactor (`reactor_new` · `r_read` · `r_write`) — take a capability as their first operand: `call_builtin net_send <capability> <handle> <bytes>`.

1c-1 Builtin ops that take no capability are **computations** — floating-point functions (§6.3.9), the sum of a piece (§6.3.7.1), and hashing · cryptography · carry-less multiplication · random sequences (`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 The standard library provides ops that wrap these builtin ops (§9.3 (1a)) — `stdio` · `args` · `env` · `clock` · `tty` · `random` · `reactor` · `math` and so on. A wrapper passes its operands on as they are, so the meaning is the same.

1f **The namespace of builtin ops is separate from the author's namespace.** The word directly after `call_builtin` is always read as the name of a builtin op, and the same spelling in any other place is always read as the author's name. So the author can give a spelling the same as a builtin op to their own op · parameter · local, and can use both together in one unit. Since the place where a word stands decides which it is, a spelling never has two meanings in one place.

> **In plain words**
>
> Builtin ops are **not meant for ordinary programs to use.** They are doors the processor selects and opens for what cannot be written as a library — computations that go down to machine instructions, work that reaches the outside world, things only the processor knows. Those who use these doors are the people building system libraries and the people building processors; application programs call the ops of the standard library that wrap them (§9). That is also why the namespace is separate — a library must be able to provide its wrappers under natural names such as `sha256`.

> **In plain words**
>
> Why confine them to a place? If an operation a program uses **once** enlarges everyone's vocabulary, the names a reader must memorise grow. That cost is paid not by the one who writes but by **everyone who reads**. For the same reason pipe stage names have meaning only inside `pipe` (§6.12), and type words stand only in their own places.

1d If a word called by its bare name is not among the author's names and has the spelling of a builtin op, the processor regards it as having left out `call_builtin` and refuses it (`E-BUILTIN-BARE`). Putting a word outside that closed set after `call_builtin` does not conform either (`E-BUILTIN-NAME`) — that place does not silently pass over to the author's names.

Rejected example — a builtin op cannot be called by its bare name

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 .
```

Diagnostic: `E-BUILTIN-BARE`

Example — written to fit its place, the same thing stands

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 .
```

Example — the author's `sha256` and the builtin op `sha256` stand together in one unit

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 There are three ways of keeping names whose meaning the language fixes out of the global vocabulary.

- **Head** — the name position of `call_builtin <name>` ((1c)). Builtin ops are such.
- **Prefix** — a group of ops shares the same leading part: `bitset_*` (sets of small numbers, §6.2.12) · `lane_*` and `reduce_*` (lanes) · `atomic_*` · `wrap_*` · `sat_*` · `chk_*` · `narrow_*`. Prefixed names are global names, but since the group does not occupy short words (`union` · `remove` · `select` · `any` …), the author can use those words as their own names.
- **Place** — words that have meaning only inside a fixed context. The stages and terminals of `pipe` are such (§6.12): the terminals `any` · `all` are words inside `pipe`, and the lane predicates are `lane_any` · `lane_all`. An op has one spelling. When a name is moved, the old spelling is not left behind as a synonym.

2 The arithmetic operations are `add` `sub` `mul` `div` `mod`. The spelling of the operation that gives a remainder is `mod`, and `rem` is **the word that opens a comment** (§6.1.2) — the same spelling is not used for two meanings.

2a `mod` is the remainder **of integers**. The remainder of floating-point numbers is `fmod` (§6.3.9), and using `mod` on floating-point operands is refused at translation (`E-TYPE-KIND`, 2026-11-02 — before that it passed the check and stopped during execution).

3 The comparison operations are `eq` `ne` `lt` `le` `gt` `ge`, and their result is `bool`.

3a A comparison between integers follows the same rule as arithmetic — the two operands shall be comparable by widening, and if they cannot be, translation is refused (`E-TYPE-SIGN`, §6.3.4 (1)). The processor does not secretly change signs for a comparison. To compare the **mathematical values** of two integers even when their signs or widths differ, call `cmp_eq` · `cmp_lt` · `cmp_le` of the standard library `intops` by name (§9) — `intops.cmp_lt i64. u64. a. b. .` is true when `a` is −1 and `b` is 0. Those ops take any pair of integer types (`usize` · `idiff` · 128 bits included).

4 The logical operations are `and` `or` `not`, and their operands and results are all `bool`. `and` and `or` *short-circuit* — if the first part alone decides the result, the second is not computed.

5 The slice operations are `len` `idx` (§6.2.6).

6 The conversion operations are `widen` `narrow` (§6.2.5).

7 The bit operations are in §6.3.5.

Free choice — FC-0002 — when NaN takes part in min and max

> In `min` · `max` of floating-point numbers, if one operand is NaN, the processor gives the value of the side that is not NaN, or gives NaN. Which of the two is chosen by the processor. If neither operand is NaN, the result is uniquely fixed. Machine min · max instructions answer differently at this place, so nailing it to one side would slow down what one instruction could finish.

8 `neg` flips the sign. The operand shall be **a signed integer or a floating-point number** — an unsigned integer has no negative numbers, so it is refused at translation (`E-TYPE-SIGN`). The type of the result is the type of the operand.

8a `neg` is arithmetic too. If the result does not fit its width, it stops (§6.3.4 (3)) — the smallest value of a signed type is that case (−128 of `i8`).

In plain words

> The names may look unfamiliar, but the rule is one — **words instead of symbols**. `+` is `add`, `<` is `lt` (less than), `==` is `eq` (equal). Symbols are not used for two reasons: symbols differ subtly in meaning from language to language (for example `^` is exponentiation in some places and exclusive or in others), and a notation that can be read out as words carries over as it is to voice input and reading aloud.

6.3.4 The rules of arithmetic

1 The two operands of binary arithmetic shall be **comparable by widening** (§6.2.5). If they are not comparable, translation is refused. If only their widths differ, they are computed at the wider one.

1a Even with mixed signs, it can be computed **if there is a widening that keeps the value**. It is enough that the unsigned side fits in the result type without losing its value — `u8` and `i16` can be mixed.

1b If there is no such widening, translation is refused. `u32` and `i32` have the same width, so the unsigned side cannot fit in the signed side, and they cannot be mixed.

2 The type of the result is the **wider** of the two operands. The processor does not invent a third type that is neither of the two operands.

3 Arithmetic happens **at the declared width**. If the result does not fit that width, it **traps**.

4 Dividing an integer by 0 traps. Dividing a floating-point number by 0 gives infinity, as IEEE 754 specifies.

5 `div` and `mod` are interpreted **with the declared sign**.

5a `div` **truncates the quotient towards 0** — `div -7 2 .` is −3.

5b `mod` is **the remainder of floored division**. The sign of the remainder **follows the divisor** — `mod -7 2 .` is 1, `mod 7 -2 .` is −1. So when the divisor is positive the result is always 0 or more and less than the divisor: if `0 < n` then `0 ≤ mod i n < n`, even when `i` is negative.

5c So `div` and `mod` **are not a pair.** `(div a b) × b + (mod a b)` equals `a` only when the two numbers have the same sign or divide evenly. The remainder that pairs with the quotient truncated towards 0 is written `sub a. mul div a. b. . b. . ..`

5d `mod` of the smallest value of a signed type by −1 is 0. `div` stops at the same place (the quotient is not in that width), but the remainder 0 is within that width.

5e The quotient and remainder that pair up are in the standard library `intops` (§9) — `div_floor` is the **floor quotient** that pairs with `mod` (`(div_floor a b) × b + (mod a b)` is always `a`), and `mod_trunc` is the **truncated remainder** that pairs with `div` (its sign follows the dividend — `mod_trunc -7 2` is −1). Both take the type as an argument (`intops.div_floor i64. a. b. .`), stop when dividing by 0, and `div_floor` of the smallest value of a signed type by −1 stops as `div` does. All are written in Lowent (§9.3 (5)).

In plain words

Why does `mod` follow the sign of the divisor? The place where a remainder is used is almost always **choosing a position** — the slot of a ring buffer, the slot of a hash table, the day of the week. The property needed there is «the result is 0 or more and less than the size», and only the remainder of floored division keeps it even for negative numbers. C's `%` gives −1 for `-7 % 2`, so that property breaks.

Example — even with mixed signs, a widening that keeps the value is enough

Source

```
module ex_mixsign .

rem `u8` fits in `i16` without losing its value.
fn ok_widen input a u8 . input b i16 . output i16 . do return add a. b. . . end .

rem A wider place may be chosen for the result.
fn ok_wider input a u8 . input b i16 . output i32 . do return add a. b. . . end .
```

Output

```
ok_widen(200, -100) = 100 · ok_wider(200, -100) = 100
```

Rejected example — without a widening that keeps the value it is refused

Source

```
module ex_sign .

proc p input a i32 . input b u32 . output i32 . effects none .
do
  return add a. b. . .     rem `u32` does not fit in `i32` — the widths are the same
end .
```

Diagnostic: `E-TYPE-SIGN`

Caution — Overflow does not wrap silently

In many languages adding 1 to 255 of `u8` gives 0 (wrapping). Lowent **traps**. If wrapping is needed, use **an operation written as such**, like `wrap_add` — whether wrapping was wanted or was a mistake shall remain in the source.

> **In plain words**
>
> *"It stops when it overflows"* may look inconvenient. But if computation continues with an overflowed value, that program's answer is already wrong; it merely becomes **unknown where it went wrong**. Stopping is the failure that can be fixed.

> **Note (informative)**
>
> Why traps do not hurt performance is in §6.4.6. If a contract proves the range, that check disappears at translation time — that is, **writing honestly makes it faster**.

6.3.5 Bitwise operations

1 Bit operations treat a value as **a sequence of bits**. Operands and results are integers, and signed integers can be handled too.

1′ If the two operands of `bit_and` · `bit_or` · `bit_xor` are of the same type, the result is of that type too; if one side is a literal, it is of the other side's type. The result of `bit_not` and of shifts · rotations (`shl` · `shr` · `rotl` · `rotr`) is of the type of the (left) operand. Putting `usize` · `idiff` and a fixed-width integer in one operation is refused at translation (`E-TYPE-NOMINAL`, §6.2 (3-2)) — the shift count is not covered by this rule.

1″ If the two operands of `bit_and` · `bit_or` · `bit_xor` are integers that differ only in width or sign, they follow the same rule as arithmetic (§6.3.4 (1)–(2)): if they are comparable by widening, the result is of the wider type; if not, translation is refused (`E-TYPE-SIGN`). The result of `count_ones` · `leading_zeros` · `trailing_zeros` is of the type of `a`.

1a The right shift `shr` means different things depending on the sign. For unsigned integers 0 comes into the vacated places (logical shift); for signed integers **the sign bit comes in** (arithmetic shift). So a negative number stays negative when shifted right.

2 The bitwise logical operations are `bit_and` `bit_or` `bit_xor` `bit_not`. The first three take two values and `bit_not` takes one.

3 The shifts are `shl` (left) and `shr` (right). The rotations are `rotl` and `rotr`, in which the bits pushed out come back in on the opposite side.

3a The count of a rotation may be of any integer type, and it rotates by **the remainder after dividing by the width** — rotating a type of width 8 by 9 is the same as rotating it by 1. Rotations do not stop.

4 The shift count shall be **less than** the width of the type. Otherwise it traps.

4a The count of `shl` · `shr` · `wrap_shl` · `wrap_shr` is **an unsigned integer type** (any width). Putting a value of a signed integer type as the count is refused at translation (`E-SHIFT-SIGNED`) — a negative count has no single fixed meaning. To use a signed value as the count, convert it explicitly to an unsigned type. A literal takes the type of its place, so it is used as it is (negative literals: (9)).

5 `wrap_shl` and `wrap_shr` **wrap** the shift count into the width before using it. On a type of width 8, 9 places are read as 1 place.

> **Rejected example — the shift count is an unsigned integer type**
>
> **Source**
>
> ```
> module ex_shift_signed .
>
> fn up input a u8 . input n i8 . output u8 . do return shl a. n. . . end .
> ```
>
> Diagnostic: `E-SHIFT-SIGNED`

6 The counting operations are `count_ones` (the number of 1 bits), `leading_zeros` (the number of consecutive 0s from the front) and `trailing_zeros` (the number of consecutive 0s from the back).

7 `byte_swap` reverses the order of the bytes. It is used when exchanging values with a place of different endianness.

Example — bitwise logic and inversion

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 .
```

Output

```
mask(12,10) = 8 · both(12,10) = 14 · diff(12,10) = 6 · flip(12) = 243
```

In plain words

`flip(12)` is 243 because it is `u8` — inverting `0000_1100` gives `1111_0011`, and that is 243. With a type of a different width the answer is different too. The answer of a bit operation is **always bound to the width of the type.**

Example — shifts and rotations

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 .
```

Output

```
up(3,2) = 12 · down(12,2) = 3 · spin(129,1) = 3 · back(3,1) = 129
```

In plain words

`spin(129, 1)` being 3 shows the difference between rotating and shifting. 129 is `1000_0001`, and rotating it one place left brings the leading 1 round to the end, giving `0000_0011` = 3. Shifting the same value with `shl` **throws** that 1 **away**.

Caution — A shift count of the width or more traps — a place that differs from C

Shifting a value of width 8 by 9 places stops the processor (`E-VM-SHIFT`). In C the same thing is **undefined behaviour**; the program keeps running and the machine decides the answer. Here it is a contract. If wrapping is wanted, **say it by name** — `wrap_shl` · `wrap_shr`. At width 8, 9 places are read as 1 place (`wrap_shr(12, 9) = 6`). Wrapping silently and writing that you want wrapping are different things (§1.3).

Example — the sign changes the meaning of a shift

Source

```
module ex_signed_shift .

rem Signed integer — the sign bit comes into the vacated places (arithmetic shift). The count is
an unsigned type ((4a)).
fn s_shr input a i32 . input n u32 . output i32 . do return shr a. n. . . end .

rem Unsigned integer — 0 comes into the vacated places (logical shift).
fn u_shr input a u32 . input n u32 . output u32 . do return shr a. n. . . end .

rem Bitwise logical operations handle signed integers too.
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 .
```

Output

```
s_shr(-8, 1) = -4 · u_shr(4294967288, 1) = 2147483644 · s_and(-8, 12) = 8 · s_not(0) = -1
```

In plain words

`-8` and `4294967288` are **the same bit pattern** in 32 bits. But shifting them one place right, the answers diverge — the former gives `-4`, the latter `2147483644`. Being a bit operation does not mean the type is not looked at. **The sign decides the meaning.**

Example — counting and byte reversal

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 .
```

Output

```
ones(7) = 3 · lead(1) = 7 · tail(8) = 3 · endian(1) = 16777216
```

In plain words

`lead(1)` being 7 is also because of the width — in `u8`, 1 is `0000_0001`, so there are seven leading zeros. `endian(1)` is 16777216 because reversing the bytes of `0x00000001` gives `0x01000000`.

8 Bit operations take **only integers** — bits are a property of machine words, so for values without that property they have no meaning. This mismatch is caught **at the place of execution** (`E-BITOP-TYPE`).

Caution —

Here is a place that cannot be caught at translation. It is written down because, if it were not, readers would **believe that translation protects them** — letting people believe that something is protected when it is not is the worst thing a norm can do (§8.6).

9 If the shift count is **proven not to be less than** the width of its type, it is refused (`E-SHIFT-RANGE`).

6.3.6 Choosing by name what to do on overflow

1 `add` · `sub` · `mul` stop on overflow (§6.3.4). If another disposition is needed, use **the op whose name states that disposition**. That is, a disposition is not a setting but **a word**.

Table 29 — Arithmetic whose disposition is chosen by name

Disposition	op	What happens on overflow
stop (default)	`add` · `sub` · `mul`	it traps — it does not go on with a wrong value
wrap	`wrap_add` · `wrap_sub` · `wrap_mul`	it wraps round within the width
saturate	`sat_add` · `sat_sub` · `sat_mul`	it stops at the end value of the width
as a value	`chk_add` · `chk_sub` · `chk_mul`	it returns the overflow **as a value** ([[option]])
no check	`div_nz`	division whose divisor is **guaranteed by its type not to be 0**. No check remains at that place

1a Arithmetic that names its overflow disposition (`wrap_*` · `sat_*` · `chk_*`) belongs to **integers**. Using it on floating-point numbers is refused at translation (`E-TYPE-KIND`) — floating-point arithmetic has an infinity for the overflowing value to go to.

2 Division by 0 is the same. At a place where the divisor is **guaranteed by its type not to be 0**, use `div_nz`, and no check remains at that place. A guaranteed divisor is one of three — an integer literal that is not 0 · the value of (2a) taken out with `some_value` · a `let` bound to that. The value `nonzero_or` of (2c) gives, and a `let` bound to it, are guaranteed divisors too. Giving `div_nz` any other divisor is refused at translation (`E-TYPE-NONZERO`). The division that stops at 0 is `div`.

2a The way to obtain that guarantee is `nonzero_of`. `nonzero_of <value>` reports **as a value** that the value is not 0 ([[option]]) — none if it is 0. Once obtained, it is not asked again at each place that divides.

> **In plain words**
>
> Instead of checking «not divided by 0» every time, **proving it once and carrying it as a value** is the same as what this language does with contracts. Removing a check is not ignoring the danger; it is confirming **in one place** that there is no danger and carrying that fact in the type.

2b To receive an overflowing carry · borrow **as a value instead of discarding it**, use the standard library `intops` (§9) — `add_carry_<width>` returns the sum and the carry (0 or 1), `sub_borrow_<width>` the difference and the borrow, and `mul_wide_<width>` the lower half `lo` and the upper half `hi` of the product, as a struct (for signed widths `mul_wide_s_<width>` — `lo` is unsigned and `hi` signed). The widths are unsigned `u8` · `u16` · `u32` · `u64` · `usize` and signed `i8` · `i16` · `i32` · `i64` · `idiff`. All are written in Lowent (§9.3 (5)).

2c `nonzero_or <value> <fallback>` gives the value when it is not 0, and the fallback when it is 0. The fallback has the value's type and must be **guaranteed** not to be 0 (a nonzero integer literal · the value of `nonzero_of` taken out with `some_value` · a `let` bound to that · the result of another `nonzero_or`) — otherwise translation is refused (`E-TYPE-NONZERO`). So the result is guaranteed not to be 0 as well: it can be the divisor of `div_nz`, and so can a `let` bound to it: `div_nz total. nonzero_or count. 1 . .`. It takes integers only (`E-TYPE-KIND`).

3 Reducing a width also chooses its disposition by name.

Table 30 — The ops that change width

op	Shape	What it does
`widen`	`widen <type> <value>`	widens. The value is preserved, so no disposition is needed

`narrow`	`narrow <type> <value>`	reduces. If it does not fit, it **stops**
`narrow_wrap`		wraps if it does not fit
`narrow_sat`		stands at the end value if it does not fit
`narrow_try`		reports **as a value** if it does not fit (⟦option⟧)

> **In plain words**
>
> Why is the disposition a name and not a build setting? As a setting, **the same source gives a different answer for each build**, and which answer it is cannot be known by reading the source. As a name, whether wrapping **was wanted or was a mistake** remains at that place — the same principle this language repeats in contracts and effects.

6.3.7 Ops that handle lanes

1 A `vec` (§6.2.11) does arithmetic per lane separately, but **crossing lanes** is done by separate ops.

Table 31 — Ops that handle lanes

op	Shape	What it does
`splat`	`splat <value>`	fills every lane with one value
`load`	`load <piece> <position>`	reads lanes from memory
`store`	`store <piece> <position> <value>`	writes lanes to memory
`load_masked` · `store_masked`	+ a mask and a default	reads or writes only the lanes whose mask is on
`lane_reverse`	`lane_reverse <value>`	reverses the order of the lanes
`lane_rotate`	`lane_rotate <value> <places>`	rotates the lanes by that much
`reduce_add` · `reduce_mul`	`reduce_add <value>`	adds · multiplies all lanes
`reduce_min` · `reduce_max`	`reduce_min <value>`	the smallest · largest of the lanes
`lane_avg`	`lane_avg <value> <value>`	the rounded average per lane
`native_lanes`	`native_lanes <type>`	asks **at translation time** how many lanes this machine handles at once

> **Caution —**
>
> The value of `lane_avg` **is fixed as (a+b+1)>>1.** If the lane is narrow, `a+b` overflows that lane, so the processor shall add in a wider place and then narrow — the two back ends of this edition do that. A processor that later wants to lower this to one machine instruction shall check that **the instruction gives exactly this value**. x86′s PAVGB/PAVGW and ARM's VRHADD/URHADD give this value, but for RISC-V's vaaddu the rounding mode depends on the vxrm register, so that cannot be said. If the same name gives different values on different machines, that drops something this language has decided to keep.

1a `splat` takes the lane count **from the declared type**. So the only place it can be used is the value position of a binding that writes a vector type: `var lim vec 4 u32 splat 5 .` .. Writing it directly inside an expression does not conform (`E-VEC-SPLAT`) — nothing in that place says the lane count.
1b `lane_avg` belongs to **integer lanes** (the value above is an integer computation). Using it on floating-point lanes is refused at translation (`E-TYPE-KIND`, 2026-11-02).

1c The ops that handle lanes take **vectors**. Giving one a single number is refused at translation (`E-TYPE-KIND`).
1d The literal of `splat <literal>` follows §6.1.4 (8)(9), taking one lane as a place of that lane type — filling floating-point lanes with an integer literal is `E-TYPE-LET`, and filling `f32` lanes with a floating-point literal beyond its range is `E-TYPE-WIDTH`. The elements of list literals (`lit vec` · `lit array` · `lit slice`) are the same.
2 An op that crosses lanes gathers as many as the lane count **into one**. The type of the result is the type of the lanes. Signed lanes are added and multiplied by sign, and if the value of `reduce_add` · `reduce_mul` does not fit that type, it stops (§6.3.4 (3)).

> **Note (informative)**
>
> Earlier, the front of this table listed `sum` and `sum_fast` as «adds all lanes». No op **did** that under those names — what adds lanes is `reduce_add`, and the name `sum` was being used in the processor for **the floating-point sum of a piece** (§6.3.7.1). One name meant two things, so the names were separated (2026-09-17).

6.3.7.1 Adding up a piece

1 There are two builtin ops that add up all the **floating-point numbers** held in a piece (⟦slice⟧), and the name says **how it adds**. The standard library `math` wraps them as ops of the same names (`math.sum_neumaier` · `math.sum_seq`).
1a `sum_neumaier <piece>` adds while **collecting the lost digits separately and giving them back.** So its error is not bound to the number of terms.
1b `sum_seq <piece>` adds once from front to back. It is fast, and its error **is proportional to** the number of terms.

> **Rejected example — returning a floating-point sum in an integer place**
>
> **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 the result is a floating-point number, but the head
> says `u64`
> end .
> ```
>
> Diagnostic: `E-TYPE-RETURN`

2 The result of both ops is **a floating-point number**. Returning it as it is to a place written as an integer is refused (`E-TYPE-RETURN`).

> **In plain words**
>
> Why does the name say the method? Adding `[1e16, 1.0, -1e16]` from front to back with `sum_seq` gives 0, and adding it with `sum_neumaier` gives 1 — neither is a wrong answer; they are **different promises**. If the name were only `sum`, the caller could not know which promise it got, and switching to the faster one would become **silently changing the answer**. Cost and error are what the name carries in this language.

> **Caution —**
>
> `native_lanes` **is not a value that changes the answer.** The lane count is part of the type (§6.2.11), so the answer is the same whether the number obtained by this question or a number written by hand is used. But choosing a number that fits the machine computes **more at once**.

6.3.8 Asking the size — `size_of`

1 `size_of <type>` gives the number of bytes one value of that type occupies, **at translation time**. The type of that number is `usize` — so are the alignment `align_of` gives and the number of lanes `native_lanes` gives.

2 It cannot be used on a type whose size cannot be known.

> **In plain words**
>
> Without this question, code that takes a type as a parameter would have to **assume the widest width**. Then a container holding one-byte things uses eight times as much. Being able to ask the size lets code that takes parameters produce its costs honestly.

6.3.9 Transcendental functions

1 The builtin ops `sin` · `cos` · `exp` · `log` · `pow` · `sqrt` · `floor` · `ceil` · `round` · `fmod` are **floating-point only**. The standard library `math` wraps them as ops of the same names (`math.sqrt x.`). Application programs call those ops.

1a `abs` takes floating-point numbers and **signed integers**. The type of the result is the type of the operand. At the smallest value of a signed integer it stops — its absolute value is not in that width (§6.3.4 (3)). Using it on an unsigned integer is refused at translation (`E-TYPE-SIGN`) — that value is already its own absolute value.

> **Rejected example — `abs` of an unsigned integer**
>
> **Source**
>
> ```
> module ex_abs_unsigned .
>
> fn f input a u32 . output u32 . do
>   return abs a. . .     rem an unsigned value is already its own absolute value
> end .
> ```
>
> Diagnostic: `E-TYPE-SIGN`

2 These require no capability and can be used **only on machines with an operating system**.

> **Open issue — M-0007 — the last bit of the results of transcendental functions**
>
> This document does not fix what value the results of `sin` · `cos` · `exp` · `log` · `pow` are, down to the last bit. But which kind of processor discretion it is (§4.3) is not marked. Reading A: leave it a free choice — write the range of choice as something like «one of the two floating-point numbers nearest the true value». Reading B: make it implementation-defined — the processor documents the math library it uses. Reading C: fix it as the correctly rounded value. The processor currently gives the value the machine's math library gives.

> **Caution —**
>
> What value the results of these functions are, down to the last bit, is not fixed by this document. The machine and the math library under it decide. If a computation depends on the last bit, use them **knowing that**.

6.3.10 A truth value is not a number

1 `and` · `or` · `not` take **only truth values**. Passing an integer is refused (`E-TYPE-LOGICAL`).
2 Places where a condition comes (§6.5) likewise take only truth values (`E-TYPE-COND`).
3 That is, there is **no** promise that 0 is false and everything else true. What is being asked shall be written at that place — as in `gt a. 0 .`.
4 A truth value cannot be an operand of arithmetic (`add` · `sub` · `mul` · `div` · `mod` · `min` · `max` · `abs` and arithmetic naming its overflow disposition) · ordering (`lt` · `le` · `gt` · `ge`) · bit operations (`bit_and` · `bit_or` · `bit_xor`) (`E-TYPE-KIND`). Two truth values are compared with `eq` · `ne`. To count a truth value as a number, write `cast u8 b. .` (§6.2.16 (4)).

In plain words

In languages that let an integer stand in a condition position, a reader **guesses from context** whether `if x` means *"x is not 0"*, *"x exists"* or *"x is true"*. A place that needs guessing is also a place where the guess goes wrong. Making people write what is asked removes that place.

Rejected example — an integer is not a condition

Source

```
module ex_cond .

fn f input a u8 . output u8 .
do
  if a. do return 1 . end .     rem write what is being asked — `gt a. 0 .`
  return 0 .
end .
```

Diagnostic: `E-TYPE-COND`

6.3.11 Widths and lanes are part of the type

1 Two `vec`s with different lane counts are different types and do not mix (`E-TYPE-LANES`).
2 Only a mask can come in a place that takes a mask (`E-TYPE-MASK`).
3 Even with the same field layout, **different names are different types** (`E-TYPE-WRAP`) — the same layout does not make the same type (§8.9).
4 Putting a value outside a narrowed range (§6.2.14) in that place is refused (`E-TYPE-RANGE`). So is writing a range that **no value can satisfy** — that is not a condition but a lie.

Rejected example — two with different lane counts are different types

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 .
```

Diagnostic: `E-TYPE-LANES`

6.3.12 The rules inside an `expr` island

1 Each piece inside an `expr` island (§6.3.2) is one form — `a.` if it is a name, a form closed by its own dot if it is a call (`expr twice a. . + 1 .`). Since a form has its own end, calls need not be wrapped in parentheses. The island is closed by its own dot.
2 Inside an island there are **only infix operators**. There is no one-place operator attached in front (`E-EXPR-UNARY`) — to flip a sign, call the op written as such (`neg a. .`).

Rejected example — there is no operator attached in front inside an island

Source

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

Diagnostic: `E-EXPR-UNARY`

In plain words

The island exists to let arithmetic be read **in the familiar order**, not to bring in a second grammar. So what can be done inside an island is narrow, and because it is narrow, the meaning inside does not waver.

6.4 Declarations and contracts

6.4.1 Declaring an op

1 An *op* is a unit of execution with a name and a contract. Ops are divided into two kinds.

pure op (fn)

An op that produces no effect at all. It always gives the same result for the same input. Since it does not touch the outside world, executing it in a different order gives the same result.

effectful op (proc)

An op that produces effects. It shall write in its own contract which effects it produces (§7.1), and if it produces more effects than written, translation is refused.

2 The kind **is always written.** There is no default — whether it is `fn` or `proc` is written in front of the name.

3 The declaration of an op consists of, in order, the name, inputs, output, effects, contract and body. Inputs are written with `input`, the output with `output`. The order is laid out **so that what comes first is used by what comes after**:

- A named promise (`satisfies`) comes directly after the name — it says first **what** this op is.
- Translation-time inputs (`comptime`) come next — the types of the following inputs and output use those names.
- Capability inputs come before data inputs — who permitted it is seen before what it receives.
- The output comes after the inputs — the output type may use the inputs' type parameters.
- Effects come after the output and before the contract — they write **what is done** with the capabilities received earlier.
- The contract is in the order input conditions (`requires`) · output promises (`ensures`) · failures (`errors`) · tests (`tests`).

3a The clauses of the head are written **in one order only**. From front to back: `satisfies` · `lowdoc` · `vector` · `priority` · `comptime` inputs · capability and region inputs (whose type is `cap …` · `region …`) · `use` · data inputs · `output` · `effects` · `link` · `variadic` · `asm` · `absorbs` · `reference` · `why` · `access` · `inplace` · `invalidates` · `parallel` · `reduce` · `requires` · `ensures` · `errors` · `tests` · `schedule`. Clauses in the same position keep the order in which they are written. The method signatures of a trait (§6.11.2) and `procs` inside an actor follow the same order. Violating this order is refused at translation (`E-CLAUSE-ORDER`). The order of inputs is also the order of the caller's arguments — so capabilities are always passed **before** data.

3b The formatter (`--fmt`) moves clauses that are not inputs, and `use` clauses, into this order. It does not move inputs among themselves — moving inputs would change the meaning of calls.

3c Each clause of the head **is closed by its own dot** — the same as statements (§6.1.6 (2)). A clause word does not close the previous clause, and neither does a newline. Meeting the next clause word or `do` without a dot is refused at translation (`E-DOT-MISSING`). The list of clauses is the head, so clauses begin with the first word after the name — even if the name has the same spelling as a clause word (`proc link …`), it is read as the name.

3d The meanings of the clause words that appear in the order of (3a), and where they are defined, are as follows.

- `satisfies` — a named promise this op keeps (§6.4.5 · §6.11.2). `lowdoc` — a description for people to read (§6.4.3).
- `vector` · `priority` — make this op an interrupt handler: the number of the interrupt it receives and its urgency (§7.7).
- `comptime` inputs — parameters fixed at translation time (§6.8). Capability inputs — the right to produce effects (§7.2).
- `use` — the allocator this op uses (§8.14).
- `link` · `variadic` — the name on the C side · arguments whose number is not fixed (§7.4).
- `asm` — the place that writes machine instructions directly (§6.9.1).
- `absorbs` · `reference` · `why` — the declaration that the spread of `unsafe` stops here, and the pure version and assumptions that ground it (§7.2.1).
- `access` · `inplace` · `invalidates` — what it touches and in which mode · writing in place · what it invalidates (§6.4.3 · §8.12).
- `parallel` · `reduce` — a loop that runs divided, and the place where the steps gather (§6.5.10).
- `requires` · `ensures` · `errors` · `tests` — the clauses of the contract (§6.4.3).
- `schedule` — comes only in the head of a test (`test`). `schedule explore_interleavings .` runs **every order** in which messages are delivered

and checks that the result is the same for each order. With `limit <number>` it runs only up to that number. If the result differs by order, the test fails (`E-SCHED-NONDET`). The meaning is specified by §6.4.13 (4).

3e **The output can be named** — `output <name> <type> .`. That name is a local of the body and **starts at 0**: `0` · `0.0` for numbers, `false` for `bool`, `none` for `option`, and for a struct the 0 of each field. The body writes to that name and returns it with `return <name> .` (as with other ops that return a value, every path shall end with `return` — §6.5.5 (3)).

- Distinguishing: there is one namespace (§6.10), so if the first word after `output` **is a type name**, the whole output is a type;

if it is not a type name and what follows is one type, that word is the name of the result.

- A type without a fixed zero (slices · `result` · `owned` · enums · newtypes · generic parameters, and structs with such fields)

cannot be named (`E-RESULT-NOZERO`). Such an output is written without a name and returns a value.

- An op without a body (`extern` · the method signatures of a trait) cannot name its result (`E-RESULT-NAMED`).
- An output that returns an array (`output [<name>] array <length> <type> .`) **has its place provided by the caller** — §6.2.6 (1c).
- `ensures` still refers to the result as `ret` (§6.4.12). The name of the result is visible only inside the body.

> **Example — a named result starts at 0**
>
> **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 .
```

Rejected example — a name given to a type without a fixed zero

Source

```
module ex_named_nozero .

fn first input xs slice u8 . output r slice u8 .
do
  return xs. .
end .
```

Diagnostic: `E-RESULT-NOZERO`

Example — declaring an op

Source

```
module ex_op .

export fn twice input n u32 . output u32 .
  requires le n. 2147483647 . .
do
  return mul n. 2 . .
end .
```

Rejected example — the output written before the input

Source

```
module ex_clause_order .

fn twice output u32 .
  input n u32 .
  requires le n. 2147483647 . .
do
  return mul n. 2 . .
end .
```

Diagnostic: `E-CLAUSE-ORDER`

Rejected example — a head clause without a dot

Source

```
module ex_clause_dot .

fn twice input n u32 output u32 .
do
  return mul n. 2 . .
end .
```

Diagnostic: `E-DOT-MISSING`

In plain words

`export` is the mark that lets other modules use this op. Without it, it is used only inside its module.

6.4.2 What a contract is

A *contract* is the **promise an op writes for itself** about its input and output. A contract is not a comment but **a statement that is checked**.

Contracts are used in three places.

a) **The processor uses them as facts** — on the premise that the contract is true, it removes checks (§6.4.5).

b) **They are enforced during execution** — a contract the processor could not prove is confirmed during execution, and if broken it traps. But whether the remaining checks are actually kept is decided by **the build mode** (§6.4.8).

c) **They speak to readers** — what must be kept to call this op is visible without opening the body.

There are two places where a contract breaks, and the diagnostic separates **whose fault it is**.

Table 32 — When a contract is broken

Clause	When it is checked	Whose fault if broken
`requires`	on entry	**the caller's** — it did not keep the condition it had to keep
`ensures`	on exit	**this op's** — it broke its own promise (the caller was told a lie)

In plain words

This distinction really pays off. When a program stops, the diagnostic says at once *"did I call it wrongly, or was that function built wrongly"*.

Caution — A contract is an asset, not a cost

It is easy to think *"writing checks makes it slower"*. In Lowent it is the opposite. When a contract is written, the processor **uses it as a fact** and removes checks during execution. If it is not written, the processor does not know, so the check remains — that is, **writing honestly is faster**.

6.4.3 The six clauses of a contract

A contract consists of the following six clauses. Not every clause is required.

precondition (requires)

A condition that shall be true at the moment of entering the op. The caller is responsible for keeping it. It is checked on entry, and if it is broken it traps.

> **postcondition** (ensures)
>
> A condition that is true when the op ends normally. The maker is responsible for keeping it. The value returned can be referred to by the name `ret`.

> **error condition** (errors)
>
> When the op fails, and the error value then. It states which error comes under which condition. The condition is read with **the values on entry** to the op — ⟦(1c)⟧.

> **effect declaration** (effects)
>
> The list of effects this op produces (§7.1). If it produces no effect at all it is `effects none`.

> **access declaration** (access)
>
> What this op touches, and how — written `access <name> <mode> .`. The modes are the seven in the table below and no more; any other word is refused (`E-CONTRACT-MODE`).

Table 33 — The modes of `access` — a closed set of seven

Mode	Enforced now	Meaning
`shared_read`	**yes**	**does not write** to this place. Writing is refused
`write_only`	**yes**	**does not read** this place. Reading is refused
`sequential`	no	goes through in order
`random`	no	touches any place
`streaming`	no	passes once and does not look again
`tiled`	no	goes through in blocks
`read_mostly`	no	mostly reads and rarely writes

> **examples** (tests)
>
> Examples that show how this op is used. The processor can run them as tests.

1a The clause `tests <name>…` writes one or more **names of ops** that test this op. The names written shall be ops visible at that place; if not, it is refused (`E-CONTRACT-UNDEF`). The name of a test block (§6.4.13) is not an op, so it cannot be written here. The `tests` clause **neither defines nor runs** tests — it **only points** at what already exists. If an op pointed at disappears, the head reports it.

1b The `lowdoc` clause writes the description of this op. One line is `lowdoc "<text>" .`; several lines are opened with `lowdoc text <end mark>`, and the text runs up to the line on which that mark stands alone. This text is read by the document output (`--doc`) and does not affect the meaning of the translated program.

1c The condition of the `errors` clause is read with **the values on entry** to the op.

1c1 If the body **never** raises that error, the processor reports it (`W-ERRORS-UNRAISED`). The clause is read as the promise «this op can fail this way», and the caller builds branches by looking at it; a branch that cannot be taken becomes **dead code that the reader cannot tell apart from live branches**. An op that passes on someone else's `result` as it is is regarded as raising that error (`try · return <call>`).

1d So the only names that can stand in that condition are **those that are the same on entry and on exit** — namely inputs that do not change and module constants. Writing in the condition a

name the body can change (an actor's state field · a module `var` · an argument in a `mut` position) is refused (`E-ERRORS-STATE`).

> **In plain words**
>
> Why on entry? In the table of ⟦(6)⟧ `errors` stands on the same side as `requires` — it is a clause that says **what the caller did wrong**. What the caller did is only **what it handed over**, so the only values that can measure that mistake are what was handed over. Read with the values on exit, a successful execution would accuse itself — an op that subtracted 30 from a balance of 50 and succeeded would, because the condition «more than the balance» is read again on exit, become *"it did not raise the error it should have"*.

> **Caution —**
>
> (1d) refuses generously. A slice in a `mut` position does not change length, so a condition that reads only `len` is in fact the same on entry and on exit, but the processor does not distinguish **which part** of the value a condition reads. Rather than pretending to distinguish what cannot be distinguished, it is better to draw the line thickly and write that down — write it with `guard` inside the body.

2 The conditions used in a contract shall be **pure expressions** — something that produces effects cannot be written in a contract.

3 `shared_read` and `write_only` are **disciplines of reading and writing**, so the processor enforces them now. That is, writing to a place declared `shared_read` is refused, and reading a place declared `write_only` is refused (`E-ACCESS-MODE`). The other five are **hints** about usage, and the processor **says** that it does not use them yet (`W-NOT-YET`). It does not silently ignore them.

4 Not being enforced does not mean they may be written carelessly. A mode is a **claim** about what this op does, and the processor may someday question that claim.

6.4.4 How to write contracts

1 Conditions are written in prefix notation. Comparison and logical operations, arithmetic, `len` and `idx` can be used.

2 If there are several `requires`, **all** shall be true.

3 A condition about **every element** of a slice is written with the following four. `elem_lt <slice> <value>` means *"every element is less than that value"*, and `elem_le` · `elem_gt` · `elem_ge` likewise.

4 These four exist because a contract **cannot write going through one by one**. A condition is a pure expression and a loop is a statement, so to speak about all the elements there has to be a word that says it.

> **Example — a contract**

Source

```
module ex_contract .

rem This op reads two bytes from a slice and makes one larger number.
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 .
```

> **In plain words**
>
> How to read the example above: *"this function shall receive a row of bytes of length 2 or more (requires), promises that the value it returns is at most 65535 (ensures), and touches nothing in the outside world (effects none)."* — this much can be known without opening the body.

6.4.5 Giving a contract a name

If several ops require the same condition, that condition can be given a name. It is written with `contract`.

An op writes that name with `satisfies` and so takes on that contract.

> **Example — a named contract**
>
> **Source**
>
> ```
> module ex_named .
>
> rem If several ops require the same condition, that condition can be given a name.
> contract nonneg do
>   requires ge a. 1 . .
> end .
>
> fn half satisfies nonneg . input a u8 . output u8 .
> do
>   return div a. 2 . .
> end .
> ```

> **In plain words**
>
> If the same condition is repeated by hand in several places, one gets fixed and another forgotten. Giving it a name makes one place to fix.

6.4.6 Contracts remove checks

The processor **takes contracts as facts** and narrows the range a value can have. If that range shows safety, it removes that check.

The checks that can be removed are these — overflow, division by 0, narrowing, slice bounds.

If a contract **is not enforced, it is not used as a fact.** That is, nothing is believed without confirmation. Believing without confirmation and removing checks is not being faster but being wrong.

The processor can report where checks it could not remove remain. A performance model that does not report is itself knowledge outside the source (§1.3).

> **Example — contracts remove checks**
>
> **Source**
>
> ```
> module ex_elim .
>
> rem Without a contract, the overflow check remains.
> export fn bare input a u8 . output u8 .
> do
>   return add a. 1 . .
> end .
>
> rem With a contract, the processor proves it and removes the check.
> export fn proven input a u8 . output u8 .
> ```

```
  requires le a. 200 . .
do
  return add a. 1 . .
end .
```

Note (informative)

The two ops have the same body. The only difference is one line of contract, and that one line removes one check during execution. This is the largest thing a contract does in this language.

Diagram — a contract becomes a fact at entry and removes the overflow check

```
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 The build mode decides the remaining checks

1 The *build mode* is written in the source as `build <mode> .`. A proved contract leaves no check in any mode. What the mode decides is the disposition of **contracts that could not be proved**.

1a The modes are **a closed four** — `debug` · `test` · `release_safe` · `release_fast`. Any other word is refused (`E-BUILD-MODE`).

2 `debug` keeps the remaining checks and, if one breaks, stops saying what broke — the kind of check that broke (a `requires`, `ensures` or `errors` clause), the name of the op that has it, and the source line of that clause. `test` says the same.

2a `test` keeps checks like `debug`. What runs tests (§6.4.13) is not this mode but **a request to run tests** — that request runs tests in any mode, and even in `test` mode an ordinary execution does not run tests. If an `expect` inside a test is false, that test fails (`E-TEST-FAIL`) — it is not the processor's error but **the claim of the person who wrote it being wrong**.

3 `release_safe` keeps the remaining checks and stops if one breaks. What it reports when it stops is **a short fixed message** naming the kind of check that broke — the contract's text (the source expression) is not put in the executable. What differs from `debug` is that, as in (4), it removes contracts of `debug` grade.

4 `release_fast` **removes** the checks of `requires` that could not be proved — both those written on the op and those of named contracts taken in with `satisfies`. `release_safe` removes the `requires` checks of `debug` grade. No mode removes the exit checks of `ensures`.

4a A contract whose checks the mode removed **is not a fact.** The processor does not remove other checks on the ground of that contract — the op runs **as if that contract had not been written**. So if a call that broke the contract overflows or goes out of range inside that op, the check of that computation stops it. This is for the same reason that `assume` is not a fact (§6.4.9).

4b The checks the processor puts at each computation — overflow · division by 0 · `MIN / -1` · indices · shift counts · narrowing · tags — are not removed in any mode. Those checks disappear only when they are proved.

5 That is, `release_fast` is the same as opening one more **trust boundary** (§4.4) — a person promises that the contracts are true. If the promise is wrong, the program runs the same as a program written without the contracts. It does not go into undefined behaviour.

Rejected example — the build modes are a closed four

Source

```
module ex_build_mode .

build nosuchmode .

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

Diagnostic: `E-BUILD-MODE`

6.4.8 Names and scopes

1 Names nest in four layers — the module, the op, the block, and blocks inside it.

2 In one scope a name points at exactly one entity. A name already alive **cannot be hidden** by declaring it again. Doing so is refused at translation.

3 The things that cannot be hidden are these four — names in the module, the names of core ops (§6.1.3), the parameter names of the same op, and local names still alive in an outer block.

4 Declaring the same name twice in the same block does not conform either. The second declaration **does not replace** the first — the same characters would merely point at two entities.

5 On leaving a block, the names declared in that block disappear. So two side-by-side blocks each declaring the same name conforms — the two names are not alive at the same time.

5a A name written in a value position must refer to something alive at that place — a local · an input · an op · a constant · a variant of an enum. If it refers to nothing, translation is refused (`E-IR-UNDEF`). The implementation names that name when it says so.

5b A local lives only **after it is declared** — using it before its declaration is refused as in (5a). A name declared in the module (op · constant · type) lives anywhere in that module regardless of declaration order: an op declared later may be called earlier.

Rejected example — a name that refers to nothing

Source

```
module ex_undeclared .

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

Diagnostic: `E-IR-UNDEF`

Rejected example — a module name cannot be hidden

Source

```
module ex_shadow .

let g u32 7 .

proc p output u32 . effects none .
do
  let g u32 1 .     rem hides the module's `g`
  return g. .
end .
```

Diagnostic: `E-NAME-SHADOW`

Rejected example — a parameter cannot be hidden

Source

```
module ex_shadow_param .

proc p input n u32 . output u32 . effects none .
do
  let n u32 1 .     rem hides the parameter `n`
  return n. .
end .
```

Diagnostic: E-NAME-SHADOW

Rejected example — the same name cannot be declared twice in the same block

Source

```
module ex_shadow_twice .

proc p output u32 . effects none .
do
  let a u32 1 .
  let a u32 2 .     rem does not replace the first
  return a. .
end .
```

Diagnostic: E-NAME-SHADOW

Rejected example — an inner block cannot hide an outer name

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 the outer `a` is still alive
    return a. .
  end .
  return a. .
end .
```

Diagnostic: E-NAME-SHADOW

6 The order of looking up names is local → module → imported modules. If imported names collide, which one is meant shall be written out.

6.4.9 Grades of contracts

1 A *grade* can be attached to one contract clause to fix **when and with what that condition is handled** — it comes directly after the clause word, as in `requires <grade> <condition> .`. The grades are the following three and no more.

grade (등급)

A mark attached to one contract clause that fixes **when, and who, is responsible for** that condition. At translation the processor proves it (`static`), at run time it is checked (`debug`), or a person promises that it is true (`assume`).

Table 34 — Grades of contracts — a closed set of three

Grade	When it is looked at	Meaning
`static`	at translation	the processor **shall prove it**. If it cannot, translation fails
`debug`	at run time	looked at only while the build mode keeps checks
`assume`	**not looked at**	only written down. The processor **does not use it as a fact**

2 If no grade is written, the processor decides — it proves what it can prove, and what it cannot it keeps as the build mode (§6.4.7) decides. That is, a grade is **a mark that overturns the default disposition**, not something needed to write a contract.

3 A condition that is `assumed` **is not used as a fact.** The processor neither keeps it nor relies on it to remove other checks — that is, that condition is **something said to the reader**.

4 This is normative because the opposite does not hold. If a condition that is not kept were taken as a fact, then when that condition is false the processor would believe it is right **while having removed checks it should not have removed**. That is, to use a contract as a fact **it shall be enforced**, and what is not enforced is not a fact.

Table 35 — The same condition, different dispositions

Written	At entry	Following checks
`requires le a 200`	checked	**removed** — because it became a fact
`requires assume le a 200`	not checked	remain — because it is not a fact

In plain words

Then what is `assume` for? It is the place to **write down** what the processor cannot prove yet but a person knows. Written down, the reader knows it, and when the processor grows later that place can be raised to `static`. Not written down, that knowledge stays only in a person's head.

Example — the same program ends differently by mode — `debug`

Source

```
module ex_mode_debug .

build debug .

fn bump input a u8 . output u8 .
  requires le a. 200 . .
do
  return add a. 1 . .
end .
```

Output

```
bump(10) = 11 · bump(250) → E-VM-CONTRACT (trap)
```

Example — the same program — `release_fast`

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 .
```

Output

```
bump(10) = 11 · bump(250) = 251 (does not stop)
```

Example — the same mode — where the contract kept the overflow away

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 .
```

Output

```
bump(10) = 11 · bump(255) → E-VM-OVERFLOW (trap)
```

Caution — `release_fast` is outside the safe subset

The same program ends differently by mode. Breaking the contract `requires le a. 200 . .` by calling with `a` as 250 stops in `debug` (`E-VM-CONTRACT`), and in `release_fast` **does not stop and gives 251.** In the third example the contract was the ground for proving that `add a. 1 .` does not overflow. When the mode removes the contract, that ground goes too, so the overflow check comes back, and `bump(255)` stops instead of giving 256 (§6.4.7 (4a)). This is one of the reasons this document limits its claim of having removed undefined behaviour to **the safe subset** (§4.4). Removing checks for speed is something that may be chosen, but **the fact of choosing shall remain in the source** — that is why the mode is not a command-line flag but a `build` statement.

6.4.10 Violations that can be decided now

1 If a call site breaks the callee's `requires` and **both sides are constants**, that violation is decided **at translation**, without waiting for execution. It is then refused (`E-CONTRACT-IMPOSSIBLE`).

2 This does not make contracts stricter; it brings forward **when the answer comes**. Before, it was translated green and caught a trap during execution.

In plain words

If a contract whose answer comes at translation is postponed until execution, that program becomes a program that **is known to be wrong only by running it**. Programs carrying a bit budget that did not fit really were shipped. If the answer comes now, it is said now.

Rejected example — both sides are constants, so it is decided now

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 is never less than 10
end .
```

Diagnostic: `E-CONTRACT-IMPOSSIBLE`

6.4.11 What cannot be a name

1 Apart from what cannot be hidden (§6.4.8), there are **characters that cannot be a name in the first place**.

Table 36 — What cannot be used as a name

What	Diagnostic	Why
keywords	`E-NAME-KEYWORD`	a keyword is a mark by which the reader **tells construct from value at a glance**
names of core ops	`E-NAME-BUILTIN`	the namespace **is flat** (no hiding), so that name would mean two things
a lone _	`E-NAME-WILDCARD`	a _ standing alone is a mark that means **anything**, not a name
names beginning with __	`E-NAME-VENDOR-RESERVED`	a place **reserved** for those who make processors and libraries
characters outside ASCII	`E-NAME-ASCII`	text · strings · `rem` comments hold any character, but **names are ASCII**

1a Here «core op» is not only those named in §6.3. Anything **called by its bare name** belongs to that set — the table of core ops in annex D is all of it. The names of builtin ops that stand after `call_builtin` (files · the network · processes · standard input and output · clocks · random numbers · the environment · terminals · floating-point functions …) are not included — they are in their own namespace (§6.3.3 (1f)). **That the names of core ops are taken is normative** — using one as a local name is refused.

> **In plain words**
>
> It may look strange to rule «names in that list cannot be used» without writing the list. But the two questions are different — **what exists** is answered by the counting side (the ledger), and **whether it occupies the namespace** is answered by the norm. The former grows and shrinks, but the latter is always true, and **only what is always true is written in this document.**

2 Two imported names **being bound to the same name** is refused too (`E-NAME-COLLISION`). Importing **does not overwrite** a name that already exists.

3 In a dotted name `<front>.<back>`, the front shall be **a declared type** (`E-NAME-QUALIFIER`). That shape puts the op in the namespace of that type; it does not point at a module.

> **In plain words**
>
> Why ASCII? This language lets text and comments hold any character while keeping **only names** narrow. Names are not only read by people but moved, attached and compared by tools, and characters that cannot be told apart by eye becoming different names makes **mistakes that cannot be found**.

> **Caution —**
>
> What reserving __ protects is not the processor but **the person writing**. Once it is fixed that that place always means «something internal», when reading someone else's code you know from the name alone that it is not yours.

6.4.12 What contracts refer to

1 The names a contract clause calls shall be **visible** at that place. Calling a name that does not exist is refused (`E-REQ-UNDEF` · `E-ENS-UNDEF`).

2 A contract that calls something the processor **cannot bring** to that place is refused too (`E-REQ-UNSUP` · `E-ENS-UNSUP`). It does not pretend to judge what it cannot judge.

3 If a named contract (§6.4.5) points at something not declared, it is refused (`E-CONTRACT-UNDEF`).

4 Declaring an error that **cannot** happen is refused (`E-CONTRACT-DEAD`) — if `requires` already filters out that case, that error branch never occurs.

4a If two `requires` on the same inputs **cannot be true together**, it is refused (`E-CONTRACT-UNSAT`) — whatever arguments it is called with, it stops at entry, so the body of that op never runs. The same place as (4): code that does not run is written in the signature.

> **In plain words**
>
> What (4) prevents is not a wrong program but **a wrong picture**. If a failure that never occurs is written in the signature, the caller writes code to handle it, and that code never runs. Code that does not run is not tested, and untested code is wrong someday.

4b A contract clause the processor **cannot enforce** reports that fact (`W-CONTRACT-IGNORED`). The entry check (`requires`) and the exit check (`ensures`) each know certain shapes; other shapes are stopped by nobody during execution, and interval analysis does not learn from them either. Such a clause stands only in documentation.

> **In plain words**
>
> (4b) pairs with (2). (2) refuses **what cannot be brought**, and (4b) reports **shapes that can be brought but cannot be set up**. Both keep the same one thing — a promise that is not checked is not left looking as if it were checked. On the `ensures` side this report did not exist until 2026-09-16, so such clauses were **silently discarded** (defect note 11).

5 If the type passed to a translation-time parameter **does not provide what that place requires**, it is refused (`E-BOUND-UNSAT`, §6.11.2).

6.4.13 Test blocks

> **test block** (시험 블록)
>
> A body without inputs or outputs, written `test <name> do … end .`. It is executed only when a request to run tests is received.

> **assertion** (단언)
>
> `expect <condition> .` — a condition a test claims to be true. If it is false, that test fails.

1 Test blocks come at the top level of a module. The head has the name and, if present, a `schedule` clause (§6.4.1 (3d)); the body has the forms that come in the body of an op. The body is checked by **the same type rules** as the body of an op — being inside a test does not let `let x u8 300 .` pass.

1a The name of a test is in **the same namespace** as the names of ops · types · modules. If tests in one module, or a test and another declaration, use the same name, it is refused (`E-NAME-DUP`, §6.4.8 (2)). Tests are reported by name and called by name, so if names overlap it cannot be said which is meant.

2 The condition of `expect <condition> .` shall be `bool`. Giving a number is refused (`E-TYPE-COND`) — this language has no rule that reads non-zero numbers as true (§6.2.16 (4)).

2a If, when tests are run, the condition of an `expect` is false, that test fails (`E-TEST-FAIL`). This is not a contract violation — a contract violation is code breaking **its own promise**; a test failure is **the claim of the person who wrote it** differing from the code. The place to fix is different.

2b The processor does not remove `expect` in any build mode (§6.4.7). A vanished assertion is a test that did not run.

2c `expect` comes only inside test blocks. Writing it in the body of an op or anywhere else is refused (`E-EXPECT-PLACE`). Promises about inputs or results are written with `requires` · `ensures` (§6.4.3), and stopping deliberately with `panic` (§7.1).

3 Tests are executed only when a request to run tests (`lowentc --test`) is received. Ordinary execution and checking (`--check`) do not run tests, and checking reports that fact (`W-TEST-NOT-RUN`) — so that passing the check is not read as passing the tests.

3a The tests of a unit that translation refuses do not run. A request to run tests also goes through the same checks as `--check` first. If a refused program reports «pass», that green is false.

3b The result of running tests is the number of tests run · the number passed · the number failed. If there are no tests at all, it says so. If even one test failed, that request ends in failure.

4 A test with the clause `schedule explore_interleavings [limit <number>] .` is run repeatedly while changing **the order** in which messages are delivered. What changes is only the order §10.5 (4b) leaves open — the order inside one actor is one (§10.5 (4a)). Without `limit` it runs up to a bound the processor decides. If it runs through every order before the bound, the result is reported as «exhaustive»; if it stops at the bound, as «up to the bound». Stopping at the bound is not reported as exhaustive.

4a If it passes in every order, the test passes. If it passes in some orders and fails in others, the test fails (`E-SCHED-NONDET`) — the result depends on the order of delivery, and the answer from running it once is only the answer for that one order. If it fails in every order, it is an ordinary test failure (`E-TEST-FAIL`).

> **Example — test blocks and assertions**

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 .
```

Rejected example — the condition of an assertion is bool — a number is not true

Source

```
module ex_test_num .

test counts do
  expect 1 .
end .
```

Diagnostic: `E-TYPE-COND`

Rejected example — two tests in one module shall have different names

Source

```
module ex_test_dup .

test same do
  expect eq 1 1 . .
end .

test same do
  expect eq 2 2 . .
end .
```

Diagnostic: `E-NAME-DUP`

Rejected example — tests and ops share one namespace

Source

```
module ex_test_ns .

fn check output bool . do return true . end .

test check do
  expect check. .
end .
```

Diagnostic: `E-NAME-DUP`

Rejected example — expect comes only inside test blocks

Source

```
module ex_expect_place .

fn half input n u8 . output u8 .
do
  expect lt n. 200 . .
  return div n. 2 . .
end .
```

Diagnostic: `E-EXPECT-PLACE`

6.5 Statements and control flow

1 A *statement* is something that is executed. An expression makes a value; a statement does work.

1a So using as a statement a form that **only gives a value and changes nothing** — a pure computation (`add s 100 .` · `eq s 7 .`) or an `fn` call — does not conform (`E-VALUE-DISCARDED`). That value goes nowhere. What was usually meant is `set s. add s. 100 . .`. A `proc` call is a statement, since its effect is its work.

> **Rejected example — a form that only gives a value cannot be used as a statement**
>
> **Source**
>
> ```
> module ex_discard .
>
> fn bump input a u64 . output u64 .
>   requires lt a. 100 . .
> do
>   var s u64 a. .
>   add s. 100 .   rem what was meant is set s (add s 100) .
>   return s. .
> end .
> ```
>
> Diagnostic: `E-VALUE-DISCARDED`

6.5.1 Statements that make names

1 `let` makes a name that **does not change**. A value once fixed does not change.

2 `var` makes a name that **can change**. Its value is changed with `set`.

2a **Parameters can be `set` too.** A parameter received by value is a **local copy** within this op, so changing it does not change the caller's value. Writing a field of a struct parameter (array fields included) is the same — it writes to the local copy. The only writes that reach the caller are `mut` · `mut_ref` positions (§8.4).

2b So what `requires` · `ensures` say through a parameter name is **the value on entry**. Even after the body `set`s that name, what the contract says is the value on entry, unchanged.

```
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 The statement that makes a name is `let <name> <type> <value> .` (`var` the same) — **the type stands after the name and before the value.** If the value does not fit that type, translation is refused. The type is not guessed: it is `let x u64 300 .`, not `let x 300 .` (`E-LET-NOTYPE`). Guessed types let bare literals slip past the width check. If the value begins with `lit`, the type is written after that `lit`, so it may be left out in front (`let p lit pt do … end . .`).

3a No word is inserted between the name and the type. The `be` of the old shape `let <name> be <type> <value> .` is not a word of this language — writing it is refused at translation (`E-LET-BE`).

3b A type ends by its characters alone: it is a constructor of the closed vocabulary (`u64` · `slice u8` · `result u64 perr` · `array 4 u8`), one name, or parentheses. User types that take arguments, and forms whose number of arguments is not fixed (such as `result u64` without an error type), are wrapped in parentheses — `let b (box u64) mk. .` · `let r (result u64) ok k. . ..`

3c **The allocator clause** stands directly after the name, before the type — `use <allocator>.` gives back at the end of the block, and `keep <allocator>.` does not give back even on leaving the block (§6.2.6 (7e)(7f) · §8.14). An allocator is a variable, so it is closed with a dot: `var v use al. option (vec`

`u32 allocs.bump_bytes) vecgen.new al. 8 . .`. The old shapes `using <allocator> be` · `using <allocator> keep be` are refused (`E-USING-OLD`). A binding whose type contains `owned` (`let n use al. option (owned node) lit node … .`) does not give back at the end of the block even when written with `use` — §8.5 (10) (11).

4 After the type **there shall be a value.** Closing without a value is refused at translation (`E-LET-NOVALUE`). This language has no way of making a name and giving it a value later.

4a The value is one form and **is closed by its own dot** (§6.1.5). After it comes the statement's closing dot, or `else` — `let n u64 find k. . .` · `let n u64 find k. . else return 0 . .`. The failure clause stands **before** the statement's closing dot: writing `else` after the statement has been closed is refused at translation (`E-STMT-ELSE`).

5 **The failure clause** — in `let <name> <type> <value> else <leave> . .`, if the type of the value is `option t` · `result t e` and the type written is the contents `t`, then when there is a value its contents are bound to the name, and when it is empty (`none` · an error) what follows `else` is executed. What follows `else` **shall leave** (`return` · `break` · `continue` · `panic`) — falling through is refused (`E-GUARD-FALLTHROUGH`). So after it the name always has contents. `var` is the same.

5c The failure clause has three shapes. `else <leave> .` is one leaving statement — that statement is closed by its own dot. `else do … end` is several statements — every path of the block shall leave. `else error <name> do … end` **binds the error value** of a `result` **to `<name>`** and looks at it inside the block — `<name>` lives only inside that block, and every path of the block shall leave. If the value is an `option`, there is no error to receive, so `else error <name>` is refused (`E-BIND-ELSE`).

5d After the failure clause comes **the statement's closing dot** — after a leaving statement too (`… else return 0 . .`), and after a block too (`… else do … end .`). If that dot is missing, translation is refused (`E-DOT-MISSING`).

5a If the value is not an `option` · `result`, or the type of the contents differs from the type written, it is refused (`E-BIND-ELSE`). The type of the value is read from the output of the op called · the declaration of the name · the output of an actor handler — for any other expression, first bind it with its type written.

5b Binding an `option` · `result` value to its contents type without `else` is refused (`E-TYPE-LET`) — a place that forgot it can fail. If stopping is acceptable, write `else panic "…" .` (the effect `panic` follows).

> **Caution —**
>
> What this rule catches is not only the mistake of leaving a blank. If `let x f64 .5 .` is written, `.5` is not a floating-point literal (§6.1.4 (2)), so that dot closes the form, and after the name **only the type remains.** A person believes they wrote a value while the processor saw none, so the diagnostic names that trap.

> **Note (informative)**
>
> It matters that the default is `let`. A name that does not change needs to be **checked only once** by the reader, but a name that can change makes them ask again at every place it is used *"what value is it here"*. That question is semantic entropy (§1.3).

> **Rejected example — no value after the type**
>
> **Source**
>
> ```
> module ex_let_novalue .
>
> fn f output u8 .
> do
>   let a u8 .        rem 0 is not put in silently
> ```

```
  return a. .
end .
```

Diagnostic: `E-LET-NOVALUE`

Rejected example — the type is not guessed

Source

```
module ex_let_notype .

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

Diagnostic: `E-LET-NOTYPE`

Rejected example — `be` is not a word of this language — tests (3a)

Source

```
module ex_let_be .

fn f output u64 .
do
  let x be u64 7 . .
  return x. .
end .
```

Diagnostic: `E-LET-BE`

Rejected example — the failure clause comes before the statement's closing dot — tests (4a)

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 .
```

Diagnostic: `E-STMT-ELSE`

Rejected example — the allocator clause is `use` · `keep` — tests (3c)

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 .
```

Diagnostic: `E-USING-OLD`

Example — the three failure clauses — tests (5)(5c)

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 .
```

Output

```
one(5) = 6 · one(0) = 0 · many(200) = 99 · why(0) = 2000 · why(500) = 1000
```

6.5.2 Conditions — `if`

Syntax

```
if-statement ::= "if" term block { "else" "if" term block } [ "else" block ] "."
block        ::= "do" { statement } "end"
```

Constraints

1 The type of the condition (`term`) shall be `bool` (§6.2.3).

2 An `if-statement` is **a statement**. It cannot come in a place where a value goes.

Dynamic semantics

3 The condition is computed in turn.

4 If its value is true, the statements of the first `block` are executed in the order written.

5 If its value is false and there is an `else if`, (3) is done again with that condition; if there is an `else`, the `block` after it is executed.

6 If its value is false and there is no `else`, nothing is executed.

Diagnostics

Table 37 — Diagnostics of the if statement

Violation	Diagnostic
(1) — the condition is not `bool`	`E-TYPE-COND`
(2) — `if` written in a value position	`E-IF-VALUE`
syntax — one more dot written after the condition closed (`if c. . do`)	`E-CLOSER-EXTRA`
syntax — no closing dot at the end of the chain	`E-DOT-MISSING`
syntax — something other than `do … end` written in the `block` position	`E-CTRL-NODO`
syntax — `else` written inside the `block`	`E-STMT-ELSE`

Examples

Example — three branches — tests (4)(5)

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 .
```

Output

```
sign(5) = 1 · sign(0) = 0 · sign(-3) = 2
```

Rejected example — `if` gives no value — tests (2)

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 set or return in each branch
  return x. .
end .
```

Diagnostic: `E-IF-VALUE`

Notes

In plain words

Why must the body be a block? The condition ends at its own dot (`if gt a. 3 . do`) — the condition never swallows the next word. Still, if the body could be written as one statement, the same `if` would have two shapes, and the moment a second statement is added, whether it is inside or outside the `if` would depend on indentation. Since the body is always `do … end`, `end` says where the body ends, and the dot after it where the statement ends.

Note (informative)

That the head's expression ends at `do` and the body is written as a block holds for `while` (§6.5.3) · the four loop heads (§6.5.3.1) · `match` (§6.6) too. If the head's expression is a value that holds a block (`if eq p. lit pt do x 1 . end . . do`), the head's body is **the last block** — for the reader's sake, wrap such a value in parentheses.

Rejected example — one more dot after the head's expression

Source

```
module ex_ctrl_dot .

fn f input a u64 . output u64 .
do
  if gt a. 3 . . do
    return 1 .
  end .
  return 0 .
end .
```

Diagnostic: `E-CLOSER-EXTRA`

Note (informative)

To get a different value per branch, `set` a name or `return` in each branch.

6.5.3 Loops — `while`

1 `while <condition> do <statements> end .` repeats the block while the condition is true. The condition is computed **on each iteration, before entering the block** — if it is false from the start, the block does not run even once.

1a The condition is one form and is closed by its own dot (`while lt i. n. . do … end .`). Writing one more dot after it is refused at translation (`E-CLOSER-EXTRA`). The statement ends at the dot after `end` (§6.1.6 (2b)).

2 `break` leaves the loop, and `continue` moves to the next iteration. Without a number, both apply to the **innermost** loop. `break` carries no value.

2b With a number after the word, they apply to the loop **counted that many outward**. `break 2 .` leaves both the innermost loop and the loop that encloses it. `continue 2 .` leaves the innermost loop and goes to the next iteration of the loop that encloses it. `1` is the same as writing no number. The number is a plain decimal literal, 1 or more, and cannot exceed the number of loops enclosing that statement (`E-LOOP-COUNT`). Loops are not given names (labels).

2c What is counted are the five loop heads (`while` · `for` · `repeat` · `range` · `cycle`). The blocks of `if` and `match` are not counted. What a loop passed on the way out opened is closed as when that loop is left normally.

> **Note (informative)**
>
> Wrapping or unwrapping a loop makes the same number point at a different loop. Then count again.

2a These two are used **only inside loops**. Writing them outside a loop does not conform (`E-LOOP-OUTSIDE`) — there is no loop to leave.

3 The condition of a loop shall be `bool` too. Writing a number in a condition position is refused at translation (`E-TYPE-COND`).

3a `while` is a statement and gives no value. It has no `else` branch (`E-STMT-ELSE`).

> **Example — leaving nested loops at once**
>
> **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 .
> ```

> **Rejected example — two loops enclose it, but it tries to leave three**
>
> **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 .
> ```
>
> Diagnostic: `E-LOOP-COUNT`

Example — conditions and loops

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 Loops that go in order — `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 The four statements repeat the block **putting the values the head decides into the name in turn**. Each head word does one thing — `for` goes over the elements of a source, `repeat` n times from 0, `range` between two ends, and `cycle` over the next values the author writes. There are no other shapes.

Table 38 — Loop heads

Head	Shape	What the name takes in turn
elements	`for x <slice> do`	the elements of the slice — from the front
element places	`for x mut <slice> do`	the slots of the slice themselves — can be read and written
counting	`repeat i <type> n do`	`0, 1, …, n−1`
range	`range i <type> a b do` · `range i <type> a b step k do`	from `a` to `b`, both ends included
recurrence	`cycle i <type> v while c next e do`	starting at `v`, then `e` on each iteration

1a Each expression of the head is one form and is closed by its own dot — `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`. The ending words (`do` · `if` · `while` · `next` · `step`) are reserved, so they cannot come as names inside an expression. Writing one more dot after a closed expression is refused at translation (`E-CLOSER-EXTRA`).

1b The old shapes `for i count <type> n . do` · `for i range <type> a b . do` · `for i be <type> v . while c . next e . do` and the filter `. where c .` are refused (E-FOR-OLD). `for` does only one job, going over elements.

2 **Elements.** The source shall be a slice. If it is not a slice, translation is refused (E-TYPE-ITER). The type of the name is the **element type** of the slice. String literals (slices of bytes) and list literals (`lit slice u16 10 20 30 .`) are sources too. If there are no elements, the block does not run even once.

2a The source ends at `do` (or at the filter's `if`) — whether it is one name (`for x xs. do`) or an expression (`for x view_array u16 b. . do`), it is the same.

3 **Element places.** In `for x mut buf. do … end .`, `x` is that slot of `buf`: reading it gives the slot's value, and `set x. v. .` writes that slot. `buf` shall be `mut` (E-TYPE-MUT). During the loop the whole of `buf` is borrowed, so reading or writing `buf` inside the block is refused at translation (E-FOR-HEAD) — the elements are touched only through `x`.

3a In `for x <slice> do` written without `mut`, `x` is a **copy** holding that iteration's element value. `x` cannot be `set` inside the block (E-FOR-HEAD) — to change the slots, write `mut` in the head.

4 **Counting.** `repeat i <type> n do` puts `0` to `n-1` into `i` in turn. If `n` is 0 or less, it does not run even once. `<type>` is the type of the count and **is an integer type that is always written** — if it is not written, or is not an integer type, translation is refused (E-FOR-STEP). `n` shall fit `<type>` (E-TYPE-WIDTH).

5 **Range.** `range i <type> a b do` goes from `a` to `b` **with both ends included**. The two ends decide the direction — `range i u64 3 5` is `3, 4, 5`, and `range i u64 5 3` is `5, 4, 3`. If the two ends are the same, it runs once. Both ends shall fit `<type>` (E-TYPE-WIDTH).

5a With `step k`, the next value is `current + k` and **the sign of `k` decides the direction.** If the start has already passed the end in that direction, it does not run even once (`range i i64 1 10 step -1`). If the next value passes the end, it ends there — `range i u8 250 255 step 3` is `250, 253`. `step 0` is refused at translation (E-FOR-STEP), and if a `k` not known at translation is 0 during execution, it stops.

5b The count **does not overflow.** `range i u8 0 255` runs 256 times and ends at 255 — it never goes beyond the type while computing what comes after the last value.

6 **Recurrence.** `cycle i <type> v while c next e do … end .` starts `i` at `v`. Before entering an iteration it looks at `c`; if true, it runs the block, and when the block ends it puts the value of `e` into `i`. If `c` is false from the start, it does not run even once. `c` shall be `bool` (E-TYPE-COND), and `v` and `e` shall fit `<type>` (E-TYPE-WIDTH). The computation of `e` is ordinary arithmetic — it stops if it overflows (§6.3.4). Both clauses `while` · `next` shall be present, in this order (E-FOR-HEAD). A `continue` inside the block **goes through** `next` — it puts the value of `e` into `i` and looks at `c` again.

7 **Filtering.** `if c` can be added at the end of the head, before `do`. It means the same as writing `if not c. . do continue . end .` at the start of the block — the condition is computed on each iteration. It attaches to any of the four head words. The `if` inside the head is not a statement but a clause — no statement can come inside the head, so they do not overlap.

8 **What the head read is frozen during the loop.** The end · `step` · source are computed **once** on entering the loop. The count name (the name of `repeat` · `range` · `cycle`) and the names the head read cannot be `set` inside the block (E-FOR-HEAD). Changing the contents of slice slots is allowed.

9 The name of a loop lives **only inside the block**. When the block ends, that name is gone — reading it after the block is refused at translation (E-NAME-SCOPE). A name that already exists in the same block cannot be used again (E-NAME-SHADOW).

10 `break` and `continue` are used inside the four loops just as in `while` (§6.5.3 (2)). None of the four gives a value.

> **Example — a head that counts numbers**

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 .
```

Output

```
the sum of the even numbers from 1 to 10
```

Example — going through a slice

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 .
```

Output

```
adds up all the elements
```

Example — element places · recurrence · filtering

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 .
```

Output

```
powers() = 7 · evens() = 20
```

Rejected example — the count name cannot be changed inside the block

Source

```
module ex_for_head .

fn f output u64 .
do
  repeat i u64 3 do
    set i. 0 .
  end .
  return 0 .
end .
```

Diagnostic: `E-FOR-HEAD`

Rejected example — `for` goes only over elements — tests (1b)

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 .
```

Diagnostic: `E-FOR-OLD`

Caution —

`in` is **not** a word of this language. Writing `for x in xs` is refused (`E-VOCAB-REMOVED`). What is gone through comes directly after the name.

6.5.4 A condition that leaves — `guard`

1 `guard` **leaves on the spot** if the condition is not true. What comes after `else` **shall leave on every path**.

0 `guard <condition> else <leave> . .` — the condition is one form and is closed by its own dot. The leaving statement after `else` is closed by its own dot too, and after it comes the dot that closes

`guard` (`guard lt i. 9 . else return 0 . .`). The same holds when what follows `else` is a block (`guard c. else do … end .`).

1a After `else` there may come one statement or a block. If it is a block, **every path** of that block shall leave — the leaving statements are `return`, `break`, `continue` and `panic`.

2 If some path of `else` does not leave but continues downward, translation is refused.

3 Once past a `guard`, its condition **is guaranteed to be true.** The processor uses that fact in the code after it.

Rejected example — `else` does not leave

Source

```
module ex_guard_bad .

proc p input n u32 . output u32 . effects none .
do
  guard le n. 5 . else set n. 0 . .   rem continues downward without leaving
  return n. .
end .
```

Diagnostic: `E-GUARD-FALLTHROUGH`

Caution — `guard` is not another name for `if not`

What `guard` says is *"if not this condition, it ends here"*. So the code below `guard` lives only in a world where the condition is true. If `else` does not leave, that promise breaks, and **the code below wrongly believes it is true** — so the language refuses it.

Example — guard

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 .
```

Example — the type is written before the value, and `guard else` may be a block

Source

```
module ex_infer_guard .

rem The type is written after the name and before the value — it is not guessed.
fn inferred output u64 .
do
  let a u64 7 .
  return a. .
end .

rem `else` may be a block. The rule is "does every path leave".
fn guarded input n u8 . output u8 .
do
  guard gt n. 5 . else do
    let x u8 1 .
```

```
    return x. .
  end .
  return 9 .
end .
```

Output

```
inferred() = 7 · guarded(3) = 1 · guarded(9) = 9
```

Rejected example — a path of `else` that does not leave is refused

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 .
```

Diagnostic: `E-GUARD-FALLTHROUGH`

6.5.5 Going back — `return`

1 `return` ends the op and returns a value. The type of the value returned shall be the same as what is written in `output`.

1a This rule is the same **wherever** `return` is — the one statement after the `else` of a `guard`, and all statements inside an `else` block, receive the same check. So an op with `output result …` returns only with `ok <value>` · `error <variant>` · an expression that gives a `result`, and returning a bare value is refused (`E-TYPE-RETURN`). Conversely, an op with a bare type written returning `ok <value>` is refused too.

Rejected example — a bare value returned in a `result` place

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 neither `ok 0` nor `error too_short`
  return ok idx b. 0 . . .
end .
```

Diagnostic: `E-TYPE-RETURN`

In plain words

This place was **outside the check** for a while (until 2026-09-25) — the `else` of a `guard` was attached as one lump, not as a list of statements, so the checker did not go down into it. So the op above passed

> translation and stopped **during execution** the moment the caller asked `is_ok`. The rarer the path (input too short), the later it shows.

1b An op whose output is named (§6.4.1 (3e)) returns that name — `return <name> .`. That name is a local that starts at 0, so there is a value to return even if the body never writes it. Another value may be returned too (the name is only a local). That every path shall end with `return` is the same as for unnamed outputs (paragraph 3).

2 An op that returns no value (`output void`) can end early by writing just `return`.

2a An op that returns no value may **run to the end of its body** without `return`. There is no value to return, so where it ends is where it goes back.

3 **An op that returns a value** shall end returning a value on every path. If there is a path that ends without a value, it does not conform.

4 The following statements are regarded as ending by returning a value — `return`, an `if` ··· `else` both of whose branches return a value, and a `match` **all of whose branches return a value**. Whether a `match` covers every case is checked separately (§6.6).

5 `guard` · `while` · `for` have paths that leave, so by themselves they are not regarded as returning a value.

6 An op whose body is machine instructions (§6.9) gives its value through `out reg`, so it is not the subject of this clause.

> **In plain words**
>
> Without this rule, on a path that ends without a value the processor would **put 0 in instead.** That 0 is nowhere in the source, so the result cannot be known even after reading the whole program (§1.3). So this language does not translate such paths at all.

> **Example — an op that gives no value may end without `return`**
>
> **Source**
>
> ```
> module ex_void .
>
> proc hold input n u8 . output void . effects none .
> do
>   let x u8 n. .
> end .
> ```

> **In plain words**
>
> There is no value to return, so **where it ends is where it goes back**. Had it been an op that returns a value, the same shape would be refused (§6.5.5 paragraph 3).

6.5.6 Blocks and indentation

1 A block is opened with `do` and closed with `end`. Indentation **has no meaning** — it is for the reader.

2 A name made inside a block is visible only inside that block. Reading that name after the block has closed does not conform (`E-NAME-SCOPE`) — on a path that did not enter, no value was ever put in that name, and the processor does not put 0 in instead (§6.5.5 (3)).

> **Rejected example — reading a name made inside a block from outside the block**
>
> **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 on the path that did not enter there is no `big`
end .
```

Diagnostic: `E-NAME-SCOPE`

In plain words

Some languages divide blocks by indentation. This language does not, because if invisible characters (spaces and tabs) change the meaning of a program, **what is seen and what is meant can diverge**. `do` and `end` are visible.

6.5.7 Three ways of saying failure

1 There are only three ways of saying failure, and their uses differ.

Table 39 — The three failure channels

What	What it says	What the caller does
`result t e`	a failure that can be fixed	asks which side and handles it. If it does not handle it, it passes it upward
`option t`	no value	asks whether there is one and takes it out, or gives a value to use instead
the `panic` effect	a contract was broken	cannot handle it. The program stops

2 The variants that go into the error type of a `result` shall be **the same** as those the op's `errors` clause writes. Returning an error not written, or writing one and not returning it, does not conform.

3 `panic` does not unwind. The program ends at the place where it stops — there is no way of catching it midway and carrying on.

Example — the three channels answer in different shapes

Source

```
module ex_channels .

enum io_error do
  too_big .
end .

rem ① a failure that can be fixed — 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 no value — 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 a broken contract — if the caller breaks the promise, it stops.
fn strict input a u8 . output u8 .
  requires le a. 200 . .
do
  return add a. 1 . .
end .
```

Output

```
halve(100) = ok 50 · halve(250) = err too_big · lookup(2) = some 20 · lookup(7) = none ·
strict(10) = 11 · strict(250) → trap
```

In plain words

The question that separates the three is *"what can the caller do"*. A missing file — another file can be tried, so it is `result`. What is searched for not being in a list is simply absent, so it is `option`. A broken contract means the caller **has already broken the promise**, and that is not something that can be fixed — so it stops.

6.5.8 Handling failure — `try` and the consuming forms

1 When calling an op that returns a `result`, `try` takes out the value on success and on failure **passes that error on upward as it is**.

2 An op that uses `try` shall itself be able to return that error. Otherwise it is refused (`E-TRY-NORESULT`) — because the failure has nowhere to go. If a tail is attached to change the channel ((4)), or it is handled **here** by the means of (3), that op need not return a `result`.

3 The ways of consuming `result` and `option` are closed to the following — `try` (including with a tail, (4)) · asking (`is_ok` `is_error` `is_some` `is_none`) · taking out (`ok_value` `error_value` `some_value`) · giving a value to use instead (`value_or`).

4 A tail can be attached after `try` to **change the channel**. The value stays the same; only the container changes.

Table 40 — Changing channels

Shape	From what to what	What is lost or gained
`try <expr> else_none`	`result` ⟶ `option`	the error **is discarded** — it no longer says why it failed
`try <expr> else_error <variant>`	`option` ⟶ `result`	**a name is given** to the absence — it says why there is none

5 Changing channels **changes the type.** The type of the place where `try <expr> else_none` is written is `option`, not the value inside it. Writing a value type and then using a channel change does not conform.

Note (informative)

This is normative because once it was not so. The processor did not see the tail word, so an op written `output u64` **returned** `some 7 .`, and still `--check` was green. If the norm does not speak about the place where the channel changes, nobody notices when a tool forgets that place.

6 Taking out is **a partial operation** (§6.2.8). Taking out the absent side traps.
7 The default of `value_or` **is evaluated only when there is no value** (lazy evaluation). So in the default position an expensive computation, or one that can fail, may be written — if there is a value, it does not run.

> **Note (informative)**
>
> `value_or some 7 . div 1 0 . .` is **7**. There is a value, so the default side does not run. That place is evaluated only when there is no value, and then the trap really happens. ☞ Until 2026-08-29 it was not so — the tool evaluated the default **first**, so a program that had a value ended in a failure it need not have had. The canon (§6.2.8 · DECISION-0003 C1) had fixed it as lazy evaluation, and the tool was fixed to match the canon. The values do not change; what changes is **which traps and effects can happen**.

> **In plain words**
>
> Writing the code that checks for failure by hand every time makes it long, and long code gets things left out. `try` reduces that repetition to one word while **leaving in the source the fact that the failure goes upward** — no failure is silently ignored.

6.5.9 Stopping — `panic`

1 `panic` stops the program at once (§5.4). It is an effect, so an op that uses it shall write that effect in its contract.
2 A pure op (`fn`) cannot use `panic`. But the trap the processor raises when a contract is broken is separate from this, and it is not something the op did.

6.5.10 Loops that run divided — `parallel`

1 Attaching the clause `parallel <piece> split .` to a loop declares that the loop may be run by several together, **dividing the piece**.
2 The processor does not believe that word but **checks it.** What it checks is that the steps of the loop do not rely on one another, and there are three conditions.

Table 41 — The conditions for a loop that can run divided

Condition	If violated	What went wrong
reads only its own share	`E-PAR-READ`	if it reads someone else's place, whether that value is still old or already new depends on **who runs first**
writes only its own share	`E-PAR-WRITE`	if two steps write the same place, the value that remains depends on the order
does not write to a place that lives across steps	`E-PAR-CARRY`	such a place **binds** the steps together. If they must gather, declare it with `reduce <accumulating place> <operation> .`

3 If `parallel` names a piece but there is no loop that runs over the length of that piece, it is refused (`E-PAR-NOLOOP`) — because it cannot be known what is to be divided.
4 The gathering operation shall **give the same answer even if the order of grouping changes**. Operations that do not are refused (`E-PAR-ASSOC`). Addition of floating-point numbers does not, so it is blocked separately (`E-PAR-FLOAT`).

4a **The starting value of the accumulating place shall be the identity of that operation.** Otherwise it is refused (E-PAR-IDENTITY). When run divided, each piece starts again from that starting value, so a value that is not the identity enters the computation **as many times as there are pieces** — that is, the divided answer differs from the answer run one by one (the promise of (2) above breaks). The identity is 0 for add · bit_or · bit_xor, 1 for mul, and 0 for max of an unsigned width.

> **In plain words**
>
> If running divided is left only as **a declaration**, then when that declaration is wrong the program usually runs fine and one day runs differently. So this language checks what is declared — if it cannot be checked, it does not simply not divide; it **refuses translation.** «Fast but sometimes wrong» is not something this language sells.

> **Caution —**
>
> **Even if it does not run divided, the answer is the same.** The conditions above are conditions for being divisible, not clauses that change the meaning. So the divided answer and the one-by-one answer are always the same — that is exactly why E-PAR-FLOAT exists.

6.5.11 What a body shall have

1 An op that says it gives a value shall give a value **on every path** (E-RETURN-PARTIAL). If any one path reaches the end without a value, it does not conform.

2 The heads of declarations (fn · proc · on · test …) shall have a do … end body (E-STMT-NODO).

3 What comes at the outermost level shall begin with **the head of a declaration** (E-TOPLEVEL).

4 If something that cannot come inside a form comes, it is refused (E-FORM-UNEXPECTED). So is a name that is not an op coming in the head position (E-HEAD-NOT-AN-OP).

> **In plain words**
>
> Without (1), the declaration «it gives a value» becomes something true **only on some paths**. The caller trusts it and uses the value, and on that path there is no value. A declaration is **a promise about every path**, so breaking it on even one path means it is not a promise.

> **Rejected example — no value on some path**
>
> **Source**
>
> ```
> module ex_partial .
>
> fn f input a u8 . output u8 .
> do
>   if gt a. 5 . do return 1 . end .
> end .                      rem on the path where `a` is 5 or less there is no value
> ```
>
> Diagnostic: E-RETURN-PARTIAL

6.5.12 Only the errors written occur

1 The errors an op raises shall be within what its errors clause writes (E-ERR-UNDECLARED).

2 An errors clause writing **a name not in** the op's error type is refused too (E-ERR-UNDEF).

3 If another op's failure is passed upward — with try <op>, or by returning that result as it is with return <op> … — that op's failure becomes this op's failure. So the errors the called op **can raise** shall be within this op's errors clause (this op's error type if there is no clause) (E-ERR-UNDECLARED).

3a The errors a called op can raise are those its errors clause writes. If there is no clause, they are **the whole error type** of that op — not writing a clause means «any error of this type can occur», and the caller reads it with exactly that meaning.

Caution —

The two rules are two sides of the same sentence — **what is written and what is raised shall be the same.** If one side overflows, the caller gets a failure it did not see; if the other side overflows, it handles a failure that will not come (for the same reason as §6.4.12 (4)).

Rejected example — raising an error not written

Source

```
module ex_err_undeclared .

enum e do
  bad .
end .

fn f output result u8 e .
do
  return error bad . .   rem `errors bad …` was not written
end .
```

Diagnostic: `E-ERR-UNDECLARED`

Rejected example — passing on someone else's failure with `return`

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` is not in `load_byte`'s promise
end .
```

Diagnostic: `E-ERR-UNDECLARED`

Rejected example — an op without a clause passes on a failure it received

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 no clause — anything of `parse_error` can occur
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` is not in `load_byte`'s promise
  return ok v. . .
end .
```

Diagnostic: `E-ERR-UNDECLARED`

6.6 Telling variants apart — `match`

1 `match` looks at which variant an `enum` value is and executes the matching block.

2 `match` **shall cover every variant.** If even one is missing, translation is refused.

3 `match` can divide not only `enum`s but integers too. Then one branch is a single value or a **range**, and a range is written `<first> to <last>` and includes both ends.

4 When dividing integers too, **every value shall be covered**. Either the ranges cover the values the type can have without gaps, or there is a branch that takes the rest (_).

5 For one branch to take several values, join them with `or`.

6 A later branch writing again what an earlier branch already covered is refused (`E-MATCH-REDUNDANT`). That branch **never runs**, and code that does not run being there is a sign that the writer's meaning and the program's meaning have parted.

7 **A bare name written in `case` is always a variant.** If it is not a declared variant, it is refused (`E-MATCH-UNDEF`). To catch anything, use _.

> ### In plain words
>
> Before 2026-09-03, a bare name that was not a variant was read as **a name that binds the value**, so a typo in a variant name, far from being caught, became **a branch that catches everything** and even satisfied the missing-branch check of (2). Adding one more variant said nothing either — that typo branch swallowed the new variant too. **A place where the processor fails to catch the easiest mistake to catch is not a rule but a hole.**

7a If more names follow a name, it is **destructuring** — unpacking a variant that carries values (§6.2.19) or binding the fields of an aggregate. The first name in that position need not be a variant (it may be the name of an aggregate).

7b If the name in `case error <name>` is a **declared variant** of that error enum, it takes only that variant. If it is not a declared variant, the whole error value is bound to that name (that branch takes every error). This applies to this place, unchanged, the discipline (7) fixes for the whole of `match` — a bare name that is a variant is a variant.

> **In plain words**
>
> Before 2026-09-16, the name in this place was **always read as a binding**. So `case error not_digit .` also took the `empty` error, a `match` that wrote only one of two variants passed the missing-branch check, and writing one per variant had the second refused as «already handled» (defect note 52). Writing a variant name and having **another variant come into that branch** is exactly the mistake (7) set out to prevent.

8 When dividing a variant that carries values (§6.2.19), the number of binding names shall **equal** the number of things that variant carries (`E-MATCH-ARITY`).

8a Putting `else do … end` in the last place of a `match` makes it **the branch that takes the rest** — it does the same as `case _ .` (the grammar of annex A.7 already writes that place). The missing-branch check counts it as covering, and writing more branches after it is caught by (6).

8b When the pattern of a branch is **nested** (`case ok (some x)` · `case ok none`), its outer tag counts as covered **when the inner pattern is itself exhaustive**. The inner being exhaustive means that the inner has _ or a binding name, or the inner wrote both `some` and `none`, or the inner branches wrote every variant of that enum.

> **In plain words**
>
> Before (8b), nested branches were **not counted at all**, so a `match` that divided exhaustively was required to have a _. A useless _ says nothing even when variants grow later, so forcing it was **making people switch off** the missing-branch check of (2).

9 For branches joined with `or` to bind values, **every branch shall have the same name in the same place** (`E-MATCH-ORBIND`). Otherwise that name would point at different things depending on which branch it came from.

> **In plain words**
>
> If (2) · (4) prevent **gaps**, (6) prevents **overlaps**. The two are the front and back of the same discipline — dividing branches is dividing the space of values **without gaps and without overlaps**, and keeping only one side is not dividing but merely laying things out.

> **Example — dividing by ranges**
>
> **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 .
> ```
>
> **Output**
>
> ```
> band(5) = 1 · band(200) = 2
> ```

Rejected example — if there are values not covered, it is refused

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 .
```

Diagnostic: `E-MATCH-INEXHAUSTIVE`

In plain words

Leaving a gap is refused at translation, so the author always has to say *"what is the rest"*. In other languages this place silently does nothing or becomes an exception, and neither can be known by reading the source.

Example — match covers every variant

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 .
```

Rejected example — a match with a variant missing

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` was not handled
  end .
end .
```

Diagnostic: `E-MATCH-INEXHAUSTIVE`

> **In plain words**
>
> The same can be done by chaining conditionals. The one advantage of `match` is **completeness** — when one more variant is made, the processor finds every place that does not handle it. A chain of conditionals does not find those places.

> **Note (informative)**
>
> Adding a new variant is common. If a person has to remember *"where all must this variant be handled"*, they will certainly miss something. `match` hands that memory over to the processor.

6.7 Tests — `test` and `expect`

1 A `test` block is the place to write the tests of that module. The processor can run it.

2 `expect` is the assertion of a test. If the condition is false, that test fails.

3 The processor **does not remove `expect` by optimisation.** A vanished test is a test that did not run.

4 Only translating does not run tests. Translation having passed does not mean *"the tests passed"*.

> **Example — a test**
>
> **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 .
> ```

> **Caution — A contract violation and a test failure are different**
>
> A contract violation is **the code breaking its own promise**, and a test failure is **a test saying that the code is wrong**. The two come out as different diagnostics — because what needs fixing differs.

6.8 Translation-time parameters — `comptime`

1 A parameter marked `comptime` **shall have its value fixed at translation time**. Types can be received that way too.

2 This is the way to use the same op for several types. The processor makes a concrete instance for each combination used.

3 That making **is visible in the source.** There are no angle brackets and no inference — which combinations were made is written at the call sites.

4 Giving a `comptime` position a value not known at translation time does not conform.

4-1 Values known at translation time are integer literals, module constants, `comptime <expr>` (the computation of §6.8.1), and local `let`s in the same op whose value is computed as in §6.8.1 (1a). `var`s and run-time values are not. A parameter used as a lane count (`vec <t> <n>`) is given only integer literals and module constants — because a concrete instance is made for each such place.

4a **Leaving empty** a `comptime` position that takes a type does not conform either (`E-MONO-NOTYPE`). It is not traced back from the types of the arguments — as (3) says, what is made shall be written at the call site.

5 A concrete instance is made for each combination used, so the number of combinations is the amount of code made. This cost can be counted at the call sites.

> **Example — receiving a translation-time value**

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 .
```

Output

```
call_it() = 7
```

> **Rejected example — a value not known at translation time cannot be given**

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 .
```

Diagnostic: `E-COMPTIME-ARG`

> **Note (informative)**

It overlaps with what generics do in other languages, but here **nothing is implicit**. What is made shall be knowable from the source alone, and that is why this language does not use inference.

6.8.1 It shall be computable at translation time

1 An expression in a `comptime` position **shall produce its value at translation**. If the current processor cannot compute it, it is refused (`E-COMPTIME-NONCONST`).

1a What the current processor computes is integer literals, module constants, local `let`s whose value came out at translation time, `machine.*`, `native_lanes <type>`, `config <name>`, and pure integer builtins (`add` · `sub` · `mul` · `div` · `mod` · `neg` · comparisons · `min` · `max` · bit operations · `shl` · `shr` · logic). A local `let`'s value coming out at translation time means that its value is a literal · one such name · `native_lanes <type>` · `comptime <expr>`. `var`s and inputs (`comptime` inputs too — the concrete instance of §6.8 is one, and the value is handed over at execution) are not computed. If a computation goes

beyond the range of values (overflow of signed 64-bit · unsigned 64-bit, division by 0, mixing unsigned values and negative numbers), that expression is not computed — an expression that would stop at execution is not turned into a different number at translation.

1c Calling a pure `fn` is computed too. The processor actually runs that `fn` at translation and puts the answer in that place. A `proc`, which can produce effects, is not run. The inputs are one integer or `bool`, slices of numbers · `bools`, structs, and slices of structs. Integer arguments shall be computable as in (1a), and slice arguments shall have their bytes fixed at translation time — quoted strings · `text` literals (as written) · files read at translation (1g) · `lit array` · `lit slice` whose elements are all constants · array or slice module constants (1d)(1e). Struct arguments shall be structs with a byte layout (§6.2.6 (9)), and either a `lit s do … end` in which each field has a value computable as in (1a) (inner structs written the same way) or a struct answer (1f) — a module constant or a call in that place. The `fn` receives that struct as it would at the boundary of the program. In a 128-bit field, write one number literal. A slice-of-structs argument is a `lit slice s …` of such literals or an array · slice answer of structs (1d)(1e) — if it is the answer of a struct with slice fields, the processor hands over the bytes those fields point at as well. `mut` inputs are not accepted. If that `fn` overflows, breaks a contract or does not end within the fixed number of steps, or an argument cannot be known at translation time, it is refused with `E-COMPTIME-NONCONST`, and the message says why.

1c-1 The fixed number of steps is a number the processor states, and the build settings change it — `build comptime_steps <n> .` in the manifest, and above that the command line `--comptime-steps <n>`. An answer run at translation can be kept and used again in the next translation; a kept answer is used only when that `fn`'s definition · input bytes · processor edition · target · step budget are all the same. With or without keeping, the answers and refusals are the same.

1d An `fn` whose answer is an array (§6.2.6 (1c)) is run too. The elements are sized numbers · `bool` · rows of such elements (arrays inside arrays) · structs (§6.2.6 (9) · (9-1)). The processor provides the place of the result, runs that `fn`, and the bytes after running become **read-only** bytes fixed in the program — the same place as a list whose elements are all constants (§6.2.6 (7) ⓑ). That answer is given a name with the same length and element type as the result of `f`, as in `let x array n t comptime f … .`. It may be a `let` inside a body or a module constant. Elements cannot be written through that name — `set` · `copy` · `swap` are refused (`E-TYPE-MUT` · `E-IMMUTABLE` for `set` on a module constant), and passing it to a `mut` position is refused (`E-TYPE-ARGMUT`). Binding it with `var`, binding it to a name of a different length or element type, or putting it in the middle of an expression is refused (`E-RESULT-PLACE`). Whether two answers with the same contents are the same place is, as in ⓑ, a **free choice** (FC-0003).

1d-1 Only pure `fns` run at translation, so such an `fn` builds no new storage — the place of the answer is provided from outside (by the processor), and slice fields inside the answer point at the `fn`'s inputs or at constant bytes of the program. The processor puts the bytes pointed at into the read-only place of (1d) and rewrites the addresses of the fields to that place. If a field points elsewhere (at bytes in that `fn`'s frame), it is refused. If the number of answers cannot be known in advance, write a generous length and count how many were filled separately (the example below).

1e An `fn` whose answer is a slice of numbers · `bools` · structs (`output slice t .`) is run too. The bytes of the answer are copied and kept as the same **read-only** bytes as (1d) — whether that `fn` gave an answer pointing at a string or a piece of its input. If the elements are structs with slice fields, the addresses of the fields are rewritten as in (1d-1). The name is given with `let x slice t comptime f … .`; it may be a `let` inside a body or a module constant, and writing is refused as in (1d). A slice of rows (`output slice (array m t) .`) is not run (`E-COMPTIME-NONCONST`) — answers of rows are given as arrays (1d).

1f An `fn` whose answer is a struct is run too. The answer is a **value** of that struct — the same as a `lit s do … end` that writes each field's run value as a constant (not a window `view s`). The fields are numbers · `bool` · arrays and slices of numbers (their bytes are put in the read-only place of (1d)) ·

structs with only such fields. A 128-bit field goes in as it is, up to its upper half. If there is a field holding `option` · `result` · a reference, it is not run (`E-COMPTIME-NONCONST`).

1g `include_text "<path>"` is the bytes (`slice u8`, read-only) of the file at that path, read at translation. The path is looked up from the folder of the source file containing this text and stays inside that folder — absolute paths and `..` are refused, and so is reaching outside that folder by following links. This shape stands only in the input position of an `fn` run at translation (1c). In other places it is refused (`E-INCLUDE-PLACE`) — because it would read as if the file were read at execution. If the file cannot be read, points outside the folder, or exceeds the size the processor states, that `comptime` place is refused saying why. Those bytes are the input bytes of (1c-1) — if the file changes, a kept answer is not used. In a build where the processor pins dependencies with a lock file, files read are also pinned **by content**, like dependencies: the entry's name is `include:<the path seen from the manifest folder>`, and a different content is refused with `E-DEP-HASH`, a missing entry with `E-DEP-UNLOCKED`. The pinning entries are written by `--lock-write`.

1b A computed answer is **a constant**. The receiving place measures that answer like a literal — if it does not fit the type of the place, it is refused like a literal (`E-TYPE-WIDTH`). The types of the names used in the answer do not become the type of the answer.

2 A `let` at module level is **a named constant**. Its value too shall come out at translation time (`E-CONST-NOTCOMPTIME`). The value is an integer literal or `comptime <expr>`. That expression can read other module constants, and it is the same whether the constant read is before or after it. If they read each other and no value comes out, it is refused.

Caution —

What (1) says is not «this expression is not a constant» but «**today's processor** cannot compute this». The two are different, and this document does not hide the difference — if what cannot be done is written as if it were impossible in principle, nobody notices when it becomes possible later.

Rejected example — no value comes out at translation

Source

```
module ex_comptime .

fn f input a u8 . output u8 .
do
  let b u8 comptime a. . .    rem `a` is fixed only at execution
  return b. .
end .
```

Diagnostic: `E-COMPTIME-NONCONST`

Example — computing a constant from constants

Source

```
module ex_comptime_names .

let KIB u64 1024 .
let PAGE u64 comptime mul 4 KIB. . . .    rem reads another module constant

fn f output u64 .
do
  let extra u64 4 .
  return comptime add PAGE. extra. . . .
end .
```

Output

```
f() = 4100
```

Example — running a pure function at translation

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 at execution it is one number
end .
```

Output

```
f() = 3628800
```

Example — building a table at translation

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 at execution it is one read-only table

fn f output u32 .
do
```

```
  return idx CRC. 255 . .
end .
```

Output

```
f() = 755167117
```

Example — trimming text at translation

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 read-only bytes "hi"

fn f output usize .
do
  return len NAME. . .
end .
```

Output

```
f() = 2
```

Example — building a message table at translation — without allocation

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 the fields point at
pieces of the input text

fn f output usize .
do
  return len field idx MSGS. 1 . en . . .
end .
```

Output

```
f() = 6
```

Example — passing a struct and receiving a slice of structs

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 a piece of the input — builds no new storage
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 .
```

Output

```
f() = 75
```

Rejected example — a file read at translation is an input of a function run at translation

Source

```
module ex_include_place .

fn f output usize .
do
  return len include_text "words.txt" . . .    rem it would read as if read at execution
end .
```

Diagnostic: `E-INCLUDE-PLACE`

Rejected example — the computed answer does not fit the receiving place

Source

```
module ex_comptime_width .

let K u64 6 .

fn f output u8 .
do
  return comptime mul K. 100 . . .    rem 600 does not fit `u8`
end .
```

Diagnostic: `E-TYPE-WIDTH`

6.9 Places that cannot be checked — `unsafe` and `extern`

1 `unsafe` marks a place that does what the language cannot check. It is **an effect**, so it shall be written in the contract (§7.1), and it spreads upward to the callers.

2 `extern` marks a place that meets something written in another language (§7.4).

3 These two places **shall be narrow**. Widening them widens by as much the area the language's guarantees do not reach.

In plain words

The word «unsafe» does not mean *"anything goes"*. It is a mark that says **from here on, not the language but a person is responsible**, and that mark remains in the source so it can be found later.

6.9.1 Writing machine instructions directly — `asm`

1 The body of an op can be written as **machine instructions**. Write an `asm <machine> .` clause in the declaration, and in the body write that machine's assembly.

2 The processor **cannot read** what is inside. The assembler reads it. That is, every invariant this language keeps can be broken inside, and the processor cannot see that. What cannot be seen **is confined** — **all** of the following four shall be present.

Table 42 — What an op that uses `asm` shall have

To have	Why
the `unsafe` mark	it marks honestly what cannot be seen
`cap machine`	it **receives the right** to issue machine instructions — not an authority floating around
`effects unsafe`	carried in the signature, so **the caller knows**
the machine name	assembly is not portable. The processor does not pretend it is

3 The machine name comes from the closed set of §5.6. Other names are refused (`E-ASM-TARGET`), and assembly for a machine **different** from the one being built for is refused too — the processor neither silently drops that op nor hands another machine's instructions to the assembler.

4 Operands are `<register> <name>` for inputs and `out <register> <name>` for outputs. Writing only a register without what to carry is refused (`E-ASM-OPERAND`).

5 The `clobber` clause writes **what** these instructions **destroy**, and the `options` clause the promises given to the processor.

> **Caution —**
>
> `asm` is a clause word, so it cannot come in a value position. That is why the name of the capability is not `cap asm` but `cap machine` — a name collision is not discovered and fixed but **removed by design**.

6 The body is **the assembly itself** — one text literal, and nothing else can come there (`E-ASM-BODY`). Machine instructions and lowered code cannot be mixed.

7 The processor cannot read inside the template, but **the operand list and the `{name}`s of the template are two expressions that say the same thing**, so whether they match is checked **both ways**.

Table 43 — Places where `asm` is refused

Diagnostic	What went wrong
`E-ASM-UNBOUND`	the `asm` clause **never declared** a name the template calls. The assembler refuses it, or worse, **simply reads whatever register was there**
`E-ASM-UNUSED`	the template **does not use** a declared operand. It burns a register and says something the assembly does not — an unchecked duplicate rots into a lie (⟦§1.1⟧)
`E-ASM-BIND`	an input operand points at something that is **not a value** (a name that is neither a parameter nor a local). If it is not a value, whatever was in that place is carried
`E-ASM-BODY`	the body is not one text literal
`E-ASM-OPERAND`	only a register was written, without what to carry
`E-ASM-NOUNSAFE` · `E-ASM-NOCAP` · `E-ASM-NOEFFECT`	one of the three things to have in the table above is missing
`E-ASM-TARGET` · `E-ASM-TARGET-UNKNOWN`	the machine name differs from the machine being built for, or is outside the closed set of ⟦§5.6⟧
`E-ASM-OPTLIE`	`options pure` **disagrees with the effects line**

8 `options pure` says *"this assembly has no side effects"*, and trusting that, the processor may **remove, hoist, or run once instead of twice** these instructions. But if the effects line says it touches input/output · state · devices, **both cannot be true.** The processor then trusts `options`, so the disagreement is refused (`E-ASM-OPTLIE`).

> **Caution —**
>
> When what the processor **trusts** and what the processor **checks** differ, the dangerous side is always what it trusts. `options` is a promise that cannot be checked, so if that promise disagrees with a declaration elsewhere, it is stopped on the spot.

6.9.2 Where C is met — the boundary of `extern`

1 An op marked `extern` **has its body in C.** So writing a body on this side is refused (`E-FFI-BODY`) — two bodies are not a program but a question nobody can answer.

1a An `extern` op whose body is in C is **a block declaration**. As a `struct` holds its fields in `do … end`, it holds its clauses in `do … end`: `extern proc <name> do <clause>* end .`. Each clause is closed by its own dot (§6.4.1 (3c)). Only clauses come in the block — if a statement comes, there are two bodies (`E-FFI-BODY`). The shape that lists clauses after the name without `do` and closes with `end` is refused (`E-STMT-NODO`) — `end` closes only its own `do` (§6.1.6 (2a)).

> **Example — an op whose body is in C — its clauses go in do … end**
>
> **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 .
> ```

> **Rejected example — clauses listed without do and closed with end**
>
> **Source**
>
> ```
> module ex_extern_nodo .
>
> unsafe extern proc c_abs input k cap c . input a i64 . output i64 .
>   effects unsafe .
>   link "llabs" .
> end
> ```
>
> Diagnostic: `E-STMT-NODO`

2 `export extern` is **the opposite direction**. C calls us, so the body is **on this side**, and without a body it is refused (`E-FFI-BODY`).

2a The name seen from C is `lw` followed, for each step of the path, by `_<length><step>`. The length is decimal and may have several digits. The path of an op in a module is [module, name], of an op belonging to a type [module, type, name], and of an op in a file without a module declaration [name] — `inside` of `module listlit` is `lw_7listlit_6inside`, and `rect.area` of `module shapes` is `lw_6shapes_4rect_4area`. A name in a module is unique whatever its kind (§6.10.3), so the path fixes the name to one, and since the length is written, underscores inside names are not read as step boundaries. Names with a digit after `lw_` belong to this rule; names with a letter after `lw_` belong to the processor itself.

2b If `link "<C name>" .` is written in the head, that name becomes the C name **as it is**. That name shall be acceptable to C — ASCII letters · digits · underscores, not starting with a digit, not a reserved

word of C or C++, and not starting with `lw_`, `__` or `_` + a capital letter. Otherwise it is refused (`E-LINK-NAME`). If two exports in one translation unit have the same C name, it is refused (`E-ABI-NAME-DUP`).

> **Rejected example — a C reserved word written as the C name**
>
> **Source**
>
> ```
> module ex_link_keyword .
>
> export fn inside input a u64 . output u64 .
>   link "inline" .
> do
>   return add a. 1 . .
> end .
> ```
>
> Diagnostic: `E-LINK-NAME`

3 An op that calls C shall have **all** of the following three. If even one is missing, it is refused.

Table 44 — What an op that calls C shall have

To have	If missing	Why
the `unsafe` mark	`E-FFI-NOUNSAFE`	inside that function everything this language keeps can be broken, and the processor cannot see it. What cannot be checked **shall at least be marked**
`input k cap c .`	`E-FFI-NOCAP`	entering C is **a right that is handed over** — the same as the heap (⟦§8.4⟧) · devices · machine instructions (⟦§6.9.1⟧). A door floating around is not a door
an effects line (at least `unsafe`)	`E-FFI-NOEFFECT`	the effects line is **how the caller learns what it takes on**. C hidden behind a clean signature is exactly the hidden cost this language sets out to remove
`link "<symbol>" .`	`E-FFI-LINK`	a C name is **a promise made to another language**, and the promise is the author's. If it were made up from the op name, the moment the op is renamed it would call another function, and that fact would be nowhere in the source

3a The right is judged **by kind**. Receiving a right other than `cap c` (allocators · file systems …) does not open this place (`E-FFI-CAPKIND`). Not having one but **what it is** makes the authority.

4 The types that cross the boundary are limited to **what C's ABI can express**. Anything else (⟦option⟧ · ⟦result⟧ · vectors · containers) is refused (`E-FFI-TYPE`). C has no such things, and the processor **does not pretend it does.**

4a **Only pieces (slices)** are mapped automatically. `slice <type>` becomes **two arguments**: «a pointer to `<type>`» and «a count». The width is known — `slice u32` is `uint32_t*`, not a byte pointer. Everything else is written by hand.

4a-1 Integer arguments cross **at their declared width** — 8 · 16 · 32-bit integers are the C integer types of that width and sign (`int8_t` · `uint8_t` · `int16_t` · `uint16_t` · `int32_t` · `uint32_t`), and other integers are `long long`. The result of an `extern` is at its declared width too. The header the processor emits writes those prototypes, so if the C side includes that header, a mismatched prototype becomes a translation error.

4b **Rights do not cross.** A `cap` is not a value but an authority, so there is nothing to hand to C.

4c A struct with a byte layout (§6.2.7 — all fields sized numbers · arrays · such structs) crosses **by value**. What is passed when calling C, and what is received and returned when C calls us, is a C struct with the same field order and widths. Whether a large struct is returned through a hidden pointer is decided by the C compiler according to the target's conventions — this language does not expose that in the signature. A struct with a 128-bit integer field does not cross (`E-INT128-PLACE`, §6.2.2 (3b-2)).

4d If an input of an op that gives a name for C to call (`export`) is an ⟦option⟧ · a vector · a bitset or a struct without a layout, or its result is one of those · a piece, the processor **does not emit** the C name and says so (`W-EXPORT-NOSYM`) — the header does not promise that name either. It never writes another type in the C signature and discards the value.

5 There is **one** ABI name, `c`. `sysv` · `win64` · `aapcs64` are not made into words — what they ask is *"what is C on this machine"*, and the machine being built for (§5.6) already fixes that answer. Adding words would make two ways of saying the same thing.

6 Variadic arguments go **one way only**. Variadic C functions can be called, but C cannot call our variadic ones — that would be work done without knowing the number and types of arguments at run time, with no place for a contract to stand.

7 There is no new convention for callbacks. An `export extern` op can already be called by C, so a callback is passing **the address of that op** (`unsafe_fn <op>`) as a value. The boundary contract is checked at the place where the door is entered.

7a So what `unsafe_fn` can point at is **only an `export extern` op**. Pointing at any other op is refused (`E-FN-NOTEXPORT`) — handing over the address of a door C cannot call is not an address but a question.

7b An op used as a callback **cannot require capabilities** (`E-FN-CAP`). What enters a callback is C, and **C has no capabilities to hand over.** Work that needs capabilities is done **outside** the callback, and inside the callback there is only computation.

7c After `unsafe_fn` there shall come **the name of an op** (`E-FN-VALUE`), and an op of that name shall actually exist (`E-FN-UNDEF`).

> **Caution —**
>
> A capability is handed over by the caller (§7.2), and the caller of a callback is C. So if a callback requires a capability, that place becomes **a place nobody can fill**. Rather than discovering it empty at run time, that shape is refused at translation.

8 Passing `owned <type>` to an `extern` **moves the responsibility to C.** This side's duty ends at that point, and giving it back afterwards is C's job. Whether that is kept **is not verified** — inside the door we are responsible, outside the door C is.

9 If a raw pointer (§8.5) mismatches at the boundary, it is refused (`E-UPTR-TYPE`).

10 **An exported type cannot take its alignment from the machine** (`E-ABI-TARGET-ALIGN`). Writing something like `align machine.cache_line` makes the layout differ by machine, while the header the C side reads assumes there is **one** layout — the two part without a word. Give exported types alignment with **a written number** (`align N .`).

> **In plain words**
>
> What this clause does is not make C convenient to call, but make **the places that call C narrow and visible**. The mark (`unsafe`) · the right (`cap c`) · the effects line each tell the same fact to people, to the processor and to the caller. With only one of the three it «can still be called» — but then this language can no longer say what it has lost.

6.9.3 Places that use instructions that differ by machine — `unsafe target`

1 Some ops have meaning **only when a particular instruction set is present**. Such ops write that fact with `unsafe target <instruction set>`.

2 An instruction-set name the processor does not know is refused (`E-TARGET-ISET`). The names form a closed set.

3 If the name is known but **the machine being built for cannot provide it**, it is refused (`E-TARGET-INTRIN`).

3a What this clause blocks is **the place where that op stands**, not each op individually. Which leaves must be inside this fence is decided only when the meaning of that leaf **depends on the machine** — an op written to give the same value on every machine is not inside this fence.

> **Note (informative)**
>
> There was a clause in this place that tied `avg` down as «machine-bound». It was deleted (2026-09-17) — §6.3.7 wrote it as **an ordinary lane op**, and both back ends of this edition emitted it not as a machine instruction but as widened integer arithmetic. That is, what it was blocking was **a machine dependency that did not even exist**. Where the text and the tool part, follow the text.

> **In plain words**
>
> This place is `unsafe` because the meaning of that op **depends on the machine**. What this language keeps shall be the same whatever the machine, and here that promise is let go — so the fact of letting go shall be written in the source.

> **Caution —**
>
> Among the library leaves that reach the host, some today go down **only to POSIX**. Trying to build such a leaf for a machine without it is refused (`E-TARGET-LEAF`) — what is not there is not sent down pretending it is.

6.9.4 Decided but not built — `raw` · `addr` · `adopt` · `clone`

1 The four words of this clause have a **decided meaning, but the implementation has not built them yet**. The implementation recognizes them, and using one is refused at translation (`E-UNSAFE-UNBUILT`) — it is not taken with another meaning or quietly dropped to the interpreter. `adopt` · `clone` · `addr` are not reserved words: if the author builds an op with that name, that op stands.

2 `raw <T>` is **a location whose storage is not known**. It is made, read, written and moved (a number added to the location) only inside an op with the `unsafe` effect. It can be copied. It can be obtained from a `ref` · `mut_ref` · slice, and then the exclusivity (§8.12) and lifetime guarantees that reference kept are given up. Writing `raw` in a type position is refused for now.

3 `addr` is **a location seen as a number** — it can only be added and compared; no value can be read or written through it. Going between it and `raw` happens only inside `unsafe`. `addr <local>` takes the location of that local. Writing `addr` in a type position, or using `addr <local>`, is refused for now.

4 `adopt` turns a `raw <T>` back into a safe slice. It requires **all** three: being inside `unsafe` · taking as a witness the live region the memory lives in (the new slice gets that region's lifetime) · stating alignment and range as a contract (`requires`). To turn it back into a writable slice, the assumption «no other access to that range» is written with `assume` — a mark that a person carries what the implementation cannot see.

5 `clone <value> <region>` **copies a value deeply** into that region. By the principle that there is no hidden copy (§8), a deep copy happens only through this word. It has the allocation effect (`alloc`).

6 **Deferred.** The four are built when a use appears. If the other paths — windows on bytes (§6.2.13) · `extern` (§6.9.2) · allocators (§8.13) — fill those uses and the four **prove unnecessary, they**

are dropped. The owner decides whether to build or drop them. Until then this clause stays and the implementation keeps recognizing the words.

> **Rejected example — a location type that is not built yet**
>
> **Source**
>
> ```
> module ex_unbuilt .
>
> fn f input p raw . output u64 . do
>   return 0 .
> end .
> ```
>
> Diagnostic: `E-UNSAFE-UNBUILT`

> **Note (informative)**
>
> The source is RFC-0005 §6.1 · §6.8 (the kinds of reference and locations ⟶ turning back into a safe reference) and RFC-0049 (`clone`). Both RFCs were adopted and then archived, and these four were not carried into the canon (sweep of 2026-11-11). The owner decided to «put them in formally, state that they are dropped if they later prove unnecessary, and keep tracking them» (2026-11-12).

6.9.5 Decided but not built: special cases — intrusive lists · graphs · a single-producer single-consumer queue · `layout c`

1 This clause writes the **special cases** of low-level data structures. As in §6.9.4, their meaning is decided but they are not yet built as words of the language. A special case is not a world of its own but **the low-level form of managed storage**: it follows the same exclusivity rule (§8.12) and the same region rules, and no borrowing rule exists only for it.

2 **Intrusive lists and graphs** — lists · trees · graphs whose nodes hold in themselves the slots that point to their neighbours. All nodes live in **one region**, and neighbours are pointed to by handles of that region, not by pointers. A type that declares such a structure **must** provide make · insert · remove · walk · release — that obligation is named `T-LINKED` in the original, and the place where it is written is the original's `manage` clause. The original left the spelling (`linked intrusive` · `graph intrusive` · `manage`) undecided, so it is decided when built. So they are not refused as words.

3 **A single-producer single-consumer queue** (the original's `queue spsc`) — a ring of fixed size touched by one writing flow and one reading flow only. It hands values over with atomic operations alone, without locks (§10). Its spelling is decided when built, for the same reason as (2).

4 **`layout c`** — lays a structure out by C's rules. The default layout already does that: fields are placed in declaration order, each at its own alignment, and the implementation does not reorder them (§6.2.15 (1)). So `layout c` was not built, and the layout words are the closed two (`packed` · `native`) — `layout c .` is refused today (`E-TYPE-LAYOUT`).

5 **Deferral.** These special cases are built when a use appears. What carries that use today: intrusive lists and node storage — the `nodelist` and `pool` libraries; the single-producer single-consumer queue — the `spsc` library; C's layout — the default layout. If it is measured that what is written as a library cannot be safe or fast without the implementation's help (types · effects · lowering), they are built; if the libraries are seen to fill the use, they are **dropped.** Whether to build or drop them is the owner's decision.

> **Rejected example — `layout c` is not built**
>
> **Source**
>
> ```
> module ex_layout_c .
> ```

```
struct header do
  layout c .
  magic u32 .
end .
```

Diagnostic: `E-TYPE-LAYOUT`

Note (informative)

The sources are RFC-0050 (low-level special cases — D4 «intrusive is not a separate paradigm» · §6.6.2 `layout c`), RFC-0051 (the hardware surface policy) and RFC-0002 §6.4.1 (`T-LINKED` · `manage`). The three were adopted and then archived, and these special cases were not carried into the canon (sweep of 2026-11-11, F6). The owner decided, as for §6.9.4, to «put them in formally, state that they are dropped if they later prove unnecessary, and keep tracking them» (2026-11-12).

6.10 Modules

1 One source file is one *module*. A module writes its own name on its first line.

2 A module name need not be the same as the file name.

6.10.1 What is exported

1 Only what is marked `export` can be used by other modules. What is not marked is used only inside that module.

2 That is, **hidden is the default**. The exporting side shall write it.

2a `export` attaches to declarations: `fn` · `proc` · `struct` · `enum` · `type` · `newtype` · `actor` · `trait`. It does not attach to module constants (`let`) · module variables (`var`) · `test` blocks (`E-TOPLEVEL`) — a constant is not part of a module's surface (give it out through an op), and a test belongs to the module that has it.

2b If the signature of an exported op uses a named type that is not exported (struct · enum · newtype · actor), the processor reports it (`W-EXPORT-HIDDEN`) — the importing side cannot write that type, so it cannot make a name to receive the value, and that export cannot be used from outside. The program itself conforms, so it is a report, not a refusal. Transparent aliases (`type`) are not included here, since the underlying type can be written. **Why it is not a refusal**: in a unit made of a single module there is no «outside», so nobody exists who cannot use that export. The processor cannot know whether this unit will be used as a library or end as a program, so it does not refuse what has not yet done harm.

Example — exporting

Source

```
module ex_export .

rem Only what is marked `export` is used by other modules.
export fn visible input n u32 . output u32 .
do
  return mul n. 2 . .
end .

rem This is used only inside this module.
fn hidden input n u32 . output u32 .
do
  return add n. 1 . .
end .
```

> **In plain words**
>
> Some languages are the other way round — public by default, with a mark to hide. This language makes hiding the default because **what is visible from outside is a promise**. Hiding something by mistake is easier to fix than promising something by mistake.

6.10.2 What is imported

1 `use` imports another module. Where to import it from is written too.
2 The things inside are called through the name of the imported module.
3 If the module is not at the place to import from, translation is refused.

> **Grammar shape — importing — the form that writes the place**

```
module <my module name> .

use <module to import> from "<the place of that module>" .
```

4 There is also a form that does not write the place. Then that module shall be **in the same translation unit** — when several files are handed over as one unit, it is looked for among them.

> **Grammar shape — importing — within the same unit**

```
module <my module name> .

use <module to import> .
```

4a `use <module> [from "<place>"] as <alias> .` **renames how that module is called.** The alias stands in every place where a module name stands — the place that qualifies an op (`<alias>.<op>`), and the place that writes a type (`<alias>.<actor>`), alike.
4b Once an alias is written, **the original module name does not stand within that unit** (`E-USE-ALIASED`). An alias is not an additional name but **a renaming**. If both names stood, there would be code that calls the same module with two spellings, and the reader would have to check whether the two are the same — the alias was written because the original name was inconvenient, and if that name kept standing, the alias would have done nothing.

> **In plain words**
>
> A discipline this language repeats stands here too: **one meaning, one spelling.** If there are two ways of writing the same thing, a reader has to know both, and someone fixing it has to fix both. The `use` line itself is the exception — that is exactly the place where the original name is written.

> **Grammar shape — importing — under another name**

```
module <my module name> .

use <module to import> as <alias> .
```

5 If a name whose place is not written is not in the unit, the processor **shall say so.** The processor of this edition says so with a warning (`W-USE-EXTERNAL`) — there is no search path for looking for modules, so whether that name exists cannot be known from this unit alone. If that name is actually used, it is refused then.

Note (informative)

Having no search path is the point of this design. If only a name were written and a tool fetched it from somewhere, what the program relies on would become knowledge outside the source — the same source could import different things on different machines. Either write the place, or hand them over together as one unit, one or the other.

Note (informative)

This place is written as **shapes** instead of examples that actually run. Importing translates only if the file actually exists, so an example that runs anywhere cannot be made. What the checker compiles is only `#ex`, and shapes are not its subject.

Note (informative)

Writing in the source where something is imported from connects with the discipline of this language — if only names were written and the tool found on its own where they come from, what the program relies on would become knowledge outside the source.

6.10.3 When names collide

1 The same name cannot be declared twice in one module. Doing so is refused at translation.

1a The top-level names of a module are in **one namespace** — modules · `fn` · `proc` (including `extern`) · `type` · `newtype` · `struct` · `enum` and its variants · `actor` · `trait` · module `let`/`var` · `test`. Even of different kinds, the same name is refused (`E-NAME-DUP`). Fields · ops belonging to a type (`<type>.<name>`) · the state and handlers of an actor · the ops a trait requires are inside their container, and in different containers the same name can be used.

2 If imported names collide with one another, which one is meant shall be written out at the place of the call.

Rejected example — the same name declared twice

Source

```
module ex_dup .

fn f output u32 . do return 1 . end .
fn f output u32 . do return 2 . end .
```

Diagnostic: `E-NAME-DUP`

Rejected example — even of different kinds it is one namespace — a module let and an fn

Source

```
module ex_dup_kinds .

let limit u64 3 .
fn limit output u64 . do return 1 . end .
```

Diagnostic: `E-NAME-DUP`

6.10.4 The order of declarations

1 The top-level declarations in a module **do not depend on order.** An op declared later can be called earlier, and ops that call each other can be used.

2 Inside an op body it is not so — local names **shall be declared before they are used**.

> **In plain words**
>
> The two rules differ because the ways of reading differ. A whole file is read by skimming, but a function body is read from top to bottom. If a name that has not appeared yet is used in a body, the reader has to go back.

6.10.5 Ops inside an op — local ops

1 `fn` · `proc` declarations can be put **directly below** an op body (in the statement positions of the body's `do … end`). These are called *local op*s. Putting them inside `if` · loops · blocks or inside another local op, or declaring types · actors · traits · `tests` · modules inside a body, does not conform (`E-LOCAL-PLACE`).

2 The name of a local op is visible **only inside the body of that op**. Inside it, order does not matter as with top-level declarations — it can be called before its declaration, and local ops of the same op can call each other and call back (§6.10.4 (1)). Other ops do not see that name. Different ops can use the same local name.

3 A local op **does not see** the outer op's parameters · locals · regions · borrows · capabilities. Using such names does not conform (`E-LOCAL-CAPTURE`). Values that are needed are passed as parameters. A local op has the same head (clauses) and body as an ordinary op, and the same rules of effects · contracts · errors · ownership. The effects of a local `proc` are the effects of the place that calls it (§7).

4 The name of a local op does not hide the name of the outer op, its parameters · locals, or names the module already has (`E-NAME-SHADOW`). Local ops of one op cannot have the same name (`E-NAME-DUP`), and the names of core ops cannot be used (`E-NAME-BUILTIN`).

5 A local op cannot carry `export` · `extern` · `unsafe` (`E-LOCAL-EXPORT`) — to use it from outside, declare it at the top level.

> **Example — a helper op put close to where it is used**

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 .
```

> **Rejected example — a local op does not see outer names**

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 .
```

Diagnostic: E-LOCAL-CAPTURE

Rejected example — a local op is put only directly below the body

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 .
```

Diagnostic: E-LOCAL-PLACE

Note (informative)

A local op is not a closure. Only the range in which its name is visible is narrow; it takes nothing secretly from outside, so the processor treats it like an ordinary op — there is no object made at run time and no hidden argument.

6.11 Type-associated ops and traits

6.11.1 Attaching an op to a type

1 If a type name and a dot are put in front of an op's name, that op becomes **attached to that type**.

2 The first input of an attached op is a value of that type.

6.11.2 Traits

1 A *trait* is **a list** of ops a type shall provide. It writes the name and signature of each op.

In plain words

When there are several shapes (rectangle · square …), the **method** of finding the area differs by type, but the **promise** *"it tells its area"* is the same. A trait gives that promise a name. Then an op that takes *"anything that tells its area"* can be written **once** — the same op need not be copied for each type, and a type that does not keep the promise is caught at translation. The call site need not know what the type is. This language has no inheritance, so this is the way to handle several types under one name.

Example — what a trait is for — one op takes several types

Source

```
module ex_trait_why .

rem Each shape finds its area differently. But the promise «it tells its area» is the same.
```

```
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 This op takes **any shape** — `requires shape t` pins it to «only types that tell their
area».
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 .
```

Output

```
demo() = 44
```

1a A trait can write **several** ops. A signature **begins with the op's name** and that op's clauses follow, and the next name opens the next signature. It is customary to write one signature per line. The clauses of a signature follow **the same order** as an op head (§6.4.1 (3a)) — inputs · output · effects. There is no limit on the number of ops written.

1b `fn` · `proc` **are not written** in a signature. What that op can do is decided by the signature's `effects` line.

- Without an `effects` line, it is an op with **no** effects. It can be provided by an `fn`, or by a `proc` that writes its effects.

A `proc` always writes an `effects` line (§7.1 without one, `E-EFFECT-MISSING`) — a `proc` that provides an effect-free signature writes `effects none` ..

- With an `effects` line, the providing side is a `proc` that writes those effects **or fewer**, or an `fn` if there are no effects.

Inside a trait, `self` refers to the type that will provide it.

If a type writes a trait with `satisfies`, the processor checks that the ops of that list exist **with exactly those signatures**. If even one is missing or mismatched, translation is refused.

There are five kinds of mismatch.

Table 45 — Places that fail to provide a trait

Diagnostic	What went wrong
`E-TRAIT-UNDEF`	`satisfies` calls a trait that **is not declared** — a claim that points at nothing checks nothing
`E-TRAIT-MISSING`	an op of the list is missing
`E-TRAIT-SIG`	the op exists but **the number of parameters** differs — a trait that does not fit exactly cannot carry a contract
`E-TRAIT-EFFECT`	the providing side has **effects the trait does not write**. The caller judges by the trait's contract, so doing what is not in that contract makes the judgment wrong
`E-TRAIT-RECV`	**an op** wrote that it provides a trait — what provides is **a type**

If an op of a trait writes `via self` in its effects line, the providing side may write **allocation-family effects** — `alloc` · `heap` · `lock` · `atomic` — for that op beyond the trait. Writing any other additional effect is still `E-TRAIT-EFFECT`. The additional effects are carried by a generic op that calls through `via <type parameter>` up to its callers (§7.1.1 (6)).

In plain words

What a trait does is make «this type knows how to do these things» **something the caller can trust**. So if only the names match and the effects differ, that trust breaks — only when the effects match too does a trait carry a contract.

Example — providing a trait

Source

```
module ex_trait .

rem A trait is the list of ops a type shall provide.
trait shape do
  area input s self . output u64 . effects none .
end .

rem Writing `satisfies` has it checked whether that list is provided.
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 .
```

Rejected example — writing that it provides and not providing

Source

```
module ex_trait_bad .

trait shape do
  area input s self . output u64 . effects none .
end .

struct rect do
  satisfies shape .     rem it wrote that it provides
  w u8 .
  h u8 .
end .

rem `rect.area` was not made
```

Diagnostic: `E-TRAIT-MISSING`

Example — a trait with several ops

Source

```
module ex_trait_many .

rem One line per signature — it begins with the name, then inputs · output · effects.
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 A signature without an effects line is provided with an `fn`.
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 A signature that writes effects is provided with a `proc` that writes those effects.
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 .
```

Rejected example — trying to provide an effect-free signature with a proc without an effects line

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 .
```

Diagnostic: `E-TRAIT-EFFECT`

In plain words

A trait is a way of writing *"this type can do these things"* **in a form that is checked**. Written as a comment, it stays the same even when the code changes, but with `satisfies`, translation is refused the moment it fails to provide.

Caution — It is not inheritance

A trait **makes no hierarchy** among types. That a type provides a trait is only the fact *"it has those ops"*; it inherits nothing from another type. This language has no inheritance.

6.11.3 Calling an attached op — method

1 `method` is the form that calls an op attached to a value. Its shape is `method <value> <step> … <name> <argument> … ..`

2 Steps and the name are told apart **by following the type**. Going down one step at a time from the receiver's type, if that type has an attached op of that name, **that is the name**, and everything after it is arguments. So `method o inner area` calls `area` attached to `inner` of `o`, and `method s plus 5` calls `plus` attached to `s` with `5`.

3 The receiver need not be a name. It may be the result of another form, and then the receiver's type is that form's output type — so `method (method x grow 2) area` holds.

4 If the receiver's type cannot be decided, the processor refuses the program. Attached ops are found **by type**, so if the type is not known, the op to call is not decided either.

Example — calling an attached op

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 .
```

> **Note (informative)**
>
> The infix notation (`s..area`) **does not exist**. It was the only place in the language read from right to left, and there was no way to write a multi-step receiver. Unified as a form, a call is **always name first**.

6.11.4 Where attached ops go wrong

1 `method` requires both **a receiver and a name** (`E-METHOD-RECV`).

2 If the receiver's type has no attached op of that name, it is refused (`E-METHOD-UNDEF`) — it shall be declared as `fn <type>.<name>`.

> **Caution —**
>
> An attached op is **a device that divides the namespace**, not inheritance (§6.11.2). So calling a name that does not exist does not search upward; it is refused right there.

> **Rejected example — that type has no attached op of that name**
>
> **Source**
>
> ```
> module ex_method_undef .
>
> struct p do
>   x u8 .
> end .
>
> fn f input s p . output u8 .
> do
>   return method s. nosuch. . .
> end .
> ```
>
> Diagnostic: `E-METHOD-UNDEF`

6.12 Flowing in one stream — `pipe`

1 `pipe` is a statement that takes the elements of a source one at a time, passes them through *stage*s in turn, and ends with one *terminal*. Its shape is as follows.

Grammar shape — the pipe statement

```
pipe <source> do
  <stage> .        rem may be absent. There may be several.
  <terminal> .     rem exactly one.
end
```

In plain words

A loop that goes through a slice almost always has the same skeleton — *keep an index, check whether it is the end, take the element out and test a condition, accumulate something, increase the index.* Writing that skeleton by hand buries **what is meant to be done** («keep only the digits and count them») between the index and the counter, and that is where off-by-one mistakes happen. `pipe` has the language take care of the skeleton and lets people write **only what is meant to be done**, one word per line. At the same time it goes through only once and makes no intermediate array, exactly like a loop written by hand (§6.12.1). The two ops below give the same answer.

Example — the same work as a hand-written loop and as a pipe — both go through once

Source

```
module ex_pipe_why .

fn is_digit input c u8 . output bool . do return and ge c. 48 . le c. 57 . . . end .

rem A hand-written loop — a person matches up the counter · index · condition · increment one by
one.
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 The same work with pipe — «keep only digits, and count». What is done shows as one word per
line.
fn digits_pipe input s slice u8 . output u64 .
do
  return pipe s. do
    filter is_digit .
    count .
  end . .
end .
```

2 There are seven stages: `filter` · `map` · `take` · `skip` · `scan` · `zip` · `enumerate`. There are five terminals: `collect into` · `fold` · `count` · `any` · `all`. Using a word outside this list in a stage position does not conform (`E-PIPE-STAGE`).

2a The place of `collect into <place>` shall be able to hold the flowing values **without losing** them. Putting wider values into a narrower container does not conform (`E-TYPE-COLLECT`) — a conversion that loses values does not happen implicitly (§6.2.5 (1)). To narrow, do it **in writing** with `map`.

2b The meaning of each word is as follows. «The flow» is the order in which the source's elements reach this place after passing the earlier stages, and «op» is a named op (4). The element type is written `t`, and the accumulator type `a`.

- `filter <op>` (stage, op is `t → bool`) — sends on only the elements for which op is true. Elements for which it is false are discarded.
- `map <op>` (stage, op is `t → u`) — calls op on each element and sends its answer on. The element type may change to `u`.
- `take <n>` (stage) — sends only the first `n` and **stops the traversal** (§6.12.2). If there are fewer than `n` elements, as many as there are.
- `skip <n>` (stage) — discards the first `n` and sends the rest.
- `enumerate <op>` (stage, op is `(u64, t) → u`) — gives the ordinal `i` (from 0) and the element to op and sends its answer.
- `zip <source2> <op>` (stage, op is `(t, t2) → u`) — gives the element and the element at the same position of `source2` to op and sends its answer. **When the shorter one ends, the flow ends.**
- `scan <initial> <op>` (stage, op is `(a, t) → a`) — sets the accumulator to `initial`, updates it for each element as `accumulator = op(accumulator, element)`, and sends the accumulator **after** the update. The initial value itself is not sent.
- `collect into <place>` (terminal) — puts the elements of the flow into `<place>` (`mut slice`) from the front and gives **the number put in** (`u64`). The place shall be long enough to hold the whole flow (2c).
- `fold <initial> <op>` (terminal, op is `(a, t) → a`) — accumulates like `scan` but gives one final accumulator (`a`). If the flow is empty, it is the initial value.
- `count` (terminal) — gives the number of elements of the flow (`u64`).
- `any <op>` (terminal, op is `t → bool`) — true if there is an element for which op is true. False if the flow is empty. It stops at the first true.
- `all <op>` (terminal, op is `t → bool`) — true if op is true for every element. True if the flow is empty. It stops at the first false.

2c When the receiving place is full, **it does not stop silently.** If it is known at translation time that the flow is longer than the receiving place — the two lengths are written in the head's contract (`requires eq len x. . N .`, which `array T N` inputs become) and no stage in between has an unknown count (`filter · zip`) — translation refuses it (`E-COLLECT-FULL`). If it cannot be known, during execution, the moment an element arrives that cannot be held, it stops saying the receiving place is full (`E-VM-BOUNDS`; natively a panic with the same sentence). To put in only as many as fit, **write** `take`.

2d The op of `filter · any · all` is **a predicate** — its output shall be `bool`. Giving an op that gives a number does not conform (`E-PIPE-PRED`). This language has no rule that reads non-zero numbers as true (§6.2.16 (4)).

> **Rejected example — putting five into three slots — both lengths are known at translation time**
>
> **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 .
> ```

Diagnostic: E-COLLECT-FULL

Rejected example — an op that gives a number given in a predicate position

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 .
```

Diagnostic: E-PIPE-PRED

3 A pipe statement has **exactly one** terminal. Ending without a terminal, or having two, does not conform.

4 The argument of a stage is **a name that points at a named op**. This language has no nameless functions (lambdas), so a function made on the spot cannot be given as a stage argument.

5 A stage op takes only one element — it has exactly one parameter and takes no capability. Using any other op as a stage does not conform (E-FOLD-OP). If a context (7) is written, there is one more parameter.

5a The op of fold and scan has two parameters, **the first being the accumulator and the second the element**. One step is accumulator = op(accumulator, element), so the type of the first parameter and the output type shall be the same. Writing them as different types does not conform (E-FOLD-ORDER).

6 The count of take and skip shall be a non-negative value fixed at translation time (§6.8). Giving a value known only at execution time does not conform.

7 Stages and terminals that call an op — filter · map · any · all · scan · fold · zip · enumerate — can have a *context* with <expr> after the op name. The context is passed as the op's **last argument**: filter · map · any · all take (element, context), fold · scan (accumulator, element, context), zip (element, partner element, context), and enumerate (ordinal, element, context). If the number of parameters or the type of the context does not match, or the context parameter is mut · owned, it does not conform (E-PIPE-CONTEXT-ARG). Writing a context on take · skip · count · collect, which call no op, or a word other than with coming after the op name, or what follows with not being one expression, does not conform (E-PIPE-WITH). with is only a marker of this place, not a keyword (annex A.1).

7a A context is **a copied value without references**. The type of the context parameter — including struct fields, the contents of every variant of an enum (unused variants too), the underlying type of aliases · newtypes, the elements of arrays, and the contents of option · result — shall not hold slices, references (ref · mut_ref), pointers, capabilities (cap), regions, owned values or actors (E-PIPE-CONTEXT-TYPE). An expression that uses a capability may **make** a context, but a capability cannot be put **inside** the context and handed over. If the context parameter is of an array type (array n t), its value crosses pointing at storage, so it cannot be a context (E-PIPE-CONTEXT-TYPE). An array that is a struct field is copied with the struct (§6.2.7 (4a)), so that struct can be a context.

7b The context expression is evaluated **once, before** the traversal, and its value is passed as the same value to every call. Even if the op changes its own copy, the next call receives the stored value. It is evaluated even if the source is empty. A pipe with a context evaluates the source and then the operands of each clause (the partner of zip, the initial value of scan · fold, the expression of with, the receiving place of collect into) in clause order, from the left. If preparation fails, the later

preparations and the traversal do not happen. The effects of the context expression are the effects of the outer op (§7). The order of a `pipe` without a context is as before — the source, the terminal's operands, then the stages in order.

Example — a threshold fixed at run time, as a context

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 .
```

Rejected example — a context is a value without references

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 .
```

Diagnostic: `E-PIPE-CONTEXT-TYPE`

Rejected example — a terminal that calls no op takes no context

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 .
```

Diagnostic: `E-PIPE-WITH`

Note (informative)

A context is not a lambda. A stage op is still a named op, and it does not secretly read outer names — it sees only the value passed as the context. References are blocked to close the way of copying only an address and watching outer storage throughout the traversal. If a large table has to be handed over, write an ordinary loop or an op call.

6.12.1 Fusion is meaning, not optimisation

1 A `pipe` statement is executed in **one pass**. No intermediate collection is made between stages. This is not an *optimisation* the processor may try but the **definition** of this statement.

2 So the processor shall not make intermediate collections. A processor that makes them does not conform.

Note (informative)

Why make it a definition? If fusion were an optimisation, *"how far is it fused"* would differ between processors, and people writing code would meet a **cliff** — the place where changing one line suddenly creates an intermediate array and makes it slow. That cliff cannot be seen from the source. Here, operations that cannot be fused **are not in the stage vocabulary at all**, so there is no way to fall off the cliff by accident.

6.12.2 Reading only as much as needed

1 `any` stops the traversal at the first element that gives true, and `all` at the first element that gives false. Elements after the stop are not read, and the stages for those elements are not executed.

2 `take n` stops the traversal after letting n elements through.

Note (informative)

This property is what lets `pipe` handle sources without an end. If stopping were not written in the meaning, a source without an end would keep the program from ever stopping.

Example — two stages and one terminal — one pass

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] → only those greater than 2 → doubled → put into out. No intermediate array is
made.
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 .
```

In plain words

pipe is a statement in order to make fusion **a promise**. Were it an expression, stages could be detached as values and passed around, and then where one stream begins and ends would not be visible in the source. do … end nails that boundary down visibly.

6.12.3 map · filter · fold outside pipe — ops of a library

1 Outside pipe, map · filter · fold **are not core ops.** The author can use the three names as the names of ops · parameters · locals. The same words inside pipe are stages and terminals (§6.12 (2)).

2 Moving a slice into another slice, filtering it or folding it into one is done by the ops map · filter · fold of the standard module collection (§9.3). **The element type brings the work along** — the element type provides the traits collection.mappable (apply) · collection.keepable (keep) · collection.foldable (combine) (§6.11.2), and the op takes that type as a translation-time argument. Giving a type that does not provide them is refused at translation.

2a collection.map and collection.filter write into the receiving place (mut slice) and give **the number of elements written** (u64) — the same as collect into. When the receiving place is full, execution stops the moment it tries to write.

2b The names that were core ops up to language revision 1.12 are six — map · filter · fold · count · seg · segs. From revision 1.13 all six are the author's names. Calling such a name without a definition is refused like any other name (E-IR-UNDEF), and the processor says where to move as well — for count, bitset_count · stack_count · len (§6.2.12 (4a)); for seg · segs, segview.piece_at · segview.piece_count; for prefix map · filter · fold, the collection of (2) above.

Example — the element type brings the work along

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 .
```

Output

```
twice_all([1,2,3], [0,0,0]) = 3
```

In plain words

These three were once core ops (in the shape `map <receiving place> <op> <source>`). The names of core ops cannot be the author's names, so three common words were blocked. This language has no way of passing an op as a value, so library generics look for the work in the element's type — the same way sorting finds comparison. Each call makes an op specific to that type, and the work is fixed as a direct call.

In plain words

Moving, filtering or folding a slice is usually written with `pipe`, without `collection` (§6.12 (2)). `pipe` does three things the way of (2) above cannot — it chooses ops by name, so one element type can have any number of transformations · predicates; it hands over values fixed at run time with `with` (§6.12 (7)); and if the op of `map` gives another type, the element type of the flow changes. Nor is there any need to wrap elements in a one-field struct. An op used in only one place is put inside that body as a local op (§6.10.5). `collection` is for general code in which the element's type has to carry the work. Calling the old prefix `map` · `filter` · `fold`, the processor too mentions `pipe` first.

Example — moving is done with pipe — run-time values go in with

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 .
```

Output

```
shifted([1,2,3], 10, [0,0,0]) = 3
```

7 Effects and capabilities

1 If §6 specified computation, this clause specifies **how that computation touches the outside world**.

7.1 Effects

1 An *effect* is an influence an op has on the outside world. Reading a file, allocating memory, waiting and the like are each an effect.

2 Effects are a fixed set of **atoms**. The list of atoms is fixed, and the author cannot make new atoms.

Table 46 — Effect atoms

Atom	What it means
`none`	no effect at all. This is the bottom
`alloc`	obtains or gives back memory from **a fixed window** — that window does not grow (⟦§8.1⟧)
`heap`	obtains memory from **a growing root** — more can be fetched during execution. Exists only on machines with an operating system (⟦§8.1⟧)
`io`	exchanges data with the outside (files · connections · standard input and output)
`wait`	waits. It wakes up someday without anyone else's help
`concurrent`	its completion **depends on the progress of another flow of execution**
`lock`	takes a lock
`atomic`	does indivisible reads and writes
`state`	changes the state a module or actor holds
`panic`	can end the program with an explicit `panic` — traps such as out-of-bounds indexing and overflow are the business of contracts, not effects (⟦§6.4.6⟧)
`device`	touches a device directly
`unsafe`	does what the language cannot check
`page_fault`	a memory access may cause a page fault
`blocking`	can hold up a flow of execution
`cancel`	can be cancelled
`detach`	detaches a flow of execution it made

2a Of these, five — `lock` · `page_fault` · `blocking` · `cancel` · `detach` — **have no core op yet** that does that work. They can be written and spread upward to the callers ((4) below), but the processor has no way of finding out whether the body does that work, so it **believes what is written**. It does not refuse them when declared but not done ((6)), and it reports that it took them on trust.

2b `device` is produced by touching device registers (`read_volatile` · `write_volatile` aimed at a device's registers, reading and writing the fields of a device register view) and by interrupt handlers (ops with a `vector` clause). Like other effects, if produced it shall be written, and if written but not produced it is reported ((6)).

3 An op shall write the effects it produces in its `effects` clause. **If it produces more effects than written, translation is refused.**

4 Effects **spread upward** to the callers. Calling an op adds that op's effects to the caller's effects.

4a `state` means changing **the state the op itself holds** (module state · the state fields of an actor). Writing to storage the caller handed over (`mut` · `mut_ref` positions, `collect into <name>`) is not `state` — that write **is already declared by the type**, the caller sees it through its own name. It is not hidden state, so there is nothing to count as an effect. Such an op writing `effects state` is allowed, and writing it is not counted as «declared but not done» (`W-EFFECT-OVER`).

5 Some effects **bring other effects along.** The following table is all of them.

Table 47 — Effects brought along (the closure relation)

If written	also counts as written	Why
`concurrent`	`wait`	if it has to wait for a peer's progress, it may be suspended

5a So a declared effect set `S` is looked at **closed** — `close(S) = S ∪ {wait ¦ concurrent ∈ S}`. Comparing two sets is also done after closing: a requirement `I` is contained in a declaration `D` when `close(I) ⊆ close(D)`. So writing `concurrent` makes writing `wait` separately unnecessary, and writing only `wait` while doing something `concurrent` is refused.

5b Closing happens **in one place only**. The table of atoms (above) stays flat, and the inclusion relation is not mixed into that table.

Diagram — effects spread upward to the callers and are compared after closing

```
 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
```

In plain words

What an effect declaration does, in one sentence — *"what can happen when this function is called is written in its signature."* It can be known without opening the body and without following the other functions it calls.

6 Conversely, **an effect declared but not produced** is a mistake too. The processor reports it. Declared effects are a cost the caller has to bear (a pure op cannot call an op that produces `io`), so writing effects that are not produced **overstates** what this op does.

7 An `fn` is an op whose purity **is fixed by contract**, so its effect set is already `none`. So **no** `effects` clause is written on an `fn`. If one is written, the processor refuses the program — the meaning is the same, but it becomes **two notations for one meaning**, and then readers and tools have to know both. `proc` is different: there the `effects` clause **narrows**. A `proc` **always** writes that clause — without it, it is `E-EFFECT-MISSING` (if there are no effects, `effects none .`). When a meaning was fixed for the absence of the clause, that meaning parted from place to place where it was read.

Example — declaring effects

Source

```
module ex_effect .

rem A pure op — it touches nothing in the outside world. `fn` itself is that declaration.
export fn double input n u32 . output u32 .
  requires le n. 1000 . .
do
  return mul n. 2 . .
```

```
end .

rem An op that produces effects — which effects is written in the contract.
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 .
```

> **Caution — An effect declaration is a check, not documentation**
>
> It differs from writing *"this function reads a file"* as a comment. A comment stays the same even when the code changes, but an effect declaration **makes translation refuse** the moment the body does more. And **doing less** is reported too — if the declaration and the body differ either way, that clause is a lie.

7.1.1 How to write the effects line

1 The effects line is **a set**, not a list. Writing the same word twice is refused (`E-EFFECT-DUP`) — the second says nothing the first did not.

2 A word not in the list is refused (`E-EFFECT-UNDEF`). The names of effects come from the closed set of §7.1.

3 `none` **cannot come together with anything else** (`E-EFFECT-NONE-MIX`). Something cannot have effects and have none at once — either write the effects or write that there are none.

4 Adding `effects none .` to an op that is already pure by contract is refused (`E-EFFECT-REDUNDANT`). One meaning has **one spelling**.

5 A pure op **cannot declare** effects (`E-EFFECT-CALC`). Being pure and having effects are different kinds, and what chooses the kind is a word, not a clause.

6 `via <type>` can be written in the effects line. It means «the **allocation-family effects** (`alloc` · `heap` · `atomic`) written by that type's ops are also this op's declaration». If a generic op writes a type parameter (`effects state via a .`), that type decides for each monomorphised instance — opened with a bump over borrowed bytes it is only `state`, and opened with an allocator standing on a growing root, `heap` stands in the instance's declaration. Effects that came in through `via` are not counted as «declared but not produced» even if this op produces only some of them.

> **In plain words**
>
> Without `via`, a generic container has to do one of two things — always write the largest effect any allocator could have (false when used with a bump), or write only the smallest (a **hidden allocation** when used with a heap). `via` lets **the type argument decide** that choice.

> **In plain words**
>
> What (1) · (4) prevent is not a wrong program but **a second way of saying the same thing**. When there are two ways, readers go looking for the difference between them and spend time finding out that there is none. And someday only one of them gets fixed.

> **Rejected example — a pure op cannot declare effects**
>
> **Source**

```
module ex_calc_eff .

fn f output u8 . effects io .     rem if it has effects, it is a `proc`
do
  return 1 .
end .
```

Diagnostic: `E-EFFECT-CALC`

7.2 Capabilities

1 To produce an effect, there shall be the right to produce it. The value that stands for that right is a *capability*.

2 Capabilities **are handed over as arguments.** There is no global capability that can be taken out and used anywhere.

3 Capabilities have kinds. Which kind is written in the type, and if the kind does not match, translation is refused. A place that requires a capability authorises **by kind** — the mere existence of a capability value is not enough.

Table 48 — Kinds of capability

Kind	What can be done with it
`cap io`	standard input and output — reading and writing
`cap clock`	asking the time
`cap random`	reading the operating system's entropy
`cap net`	making connections and exchanging data over them — **turning names into addresses belongs here too**
`cap file_system`	opening · reading · writing files and directories
`cap process`	starting other programs, reading their output, waiting for them to end or ending them
`cap tty`	asking the terminal's size and changing its mode
`cap allocator`	obtaining memory from a fixed window (it does not grow)
`cap heap`	obtaining memory from a growing root — can be received only on machines with an operating system
`cap env`	reading the variables of the execution environment
`cap c`	calling C functions (§6.9)
`cap mmio`	reading and writing device registers
`cap args`	counting · reading the arguments given when the program was called
`cap machine`	issuing machine instructions directly (⟦§6.9⟧) — the processor cannot see inside

3a The **names** of capabilities are not closed — the author can make their own capability and receive it as a parameter. What is closed is **the kinds whose meaning the processor fixes**, and those are the rows of the table above.

3b To do work whose meaning the processor fixes (asking the time · opening a connection and the like), a capability of **the kind that work requires** shall be passed. If it does not match, it is refused in one of the following two ways.

Table 49 — The two shapes of a mismatched capability

What was passed	What happens
a capability of **another kind** (`cap io` to a clock)	refused as being of a different kind
a capability with **a name the processor does not know**	refused because that effect cannot be produced with that capability

> **Note (informative)**
>
> Kinds are divided for **least authority**. If there were only one capability, a right received to read a file could also open connections, and then an op's signature would no longer say *"what this op can do"*. Because kinds are divided, the extent of the outside world that can be reached is known from the signature alone.

> **Diagram — capabilities are received at the entry point and go down only as arguments**
>
```
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 The entry point of a program receives capabilities from the execution environment (§5.5). A capability the entry point did not receive exists nowhere in the program.

5 A capability is **a mark at translation time**, not a value at run time. So an op that receives a capability does not receive one more run-time argument because of it.

> **In plain words**
>
> That is, writing `input k cap io .` is **free**. An op that receives a capability and an op that does not have **the same signature** at the C boundary — a capability asks *"is there the right to call this op"* at translation, and nothing is left to ask at execution. The point of this design is that the cost of guarding the boundary does not come back as a run-time cost. *"Making it safe makes it slow"* is not true here.

6 **Effects and capabilities are a pair.** The effects line writes *what* the op does, and the capability inputs write *who permitted it*. The same thing is written from two sides, and each checks the other:

- The work written in the effects line shall receive as an input the capability that permits it (refused if absent).
- If the body does work that a capability opens, that effect shall be written in the effects line (`E-EFFECT` if absent).
- Merely receiving a capability does not create an effect — a capability received and not used does no work.

Table 50 — Effects and the capabilities that permit them

Effect	Permitting capability	When absent
`io`	`cap io` · `cap file_system` · `cap net` · `cap tty` · `cap clock` · `cap random`	`E-EFFECT-NO-CAP`
`alloc`	`cap allocator` (or a lexical `region` block)	`E-ALLOC-NOCAP`
`heap`	`cap heap`	`E-HEAP-NOCAP`
`atomic`	`cap atomic`	`E-ATOMIC-NOCAP`

`device`	`cap mmio` — at the place that actually touches registers	`E-MMIO-NOCAP`
`state` · `panic` · `wait` · `concurrent`	— written without a capability	—

In plain words

They are a pair so that one line of signature answers two questions. From `effects io` alone one knows that *"it talks to the outside"* but not **with what**. `input fs cap file_system .` narrows that — it can open files but cannot open connections. Conversely, with only a capability and no effects line, the caller cannot know whether that op is pure.

Example — it can be produced only with a capability received

Source

```
module ex_io .

use stdio .

rem To output, `cap io` shall be received as an argument.
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 .
```

Rejected example — without a capability not one byte can be put out

Source

```
module ex_io_bad .

use stdio .

proc main output u8 . effects io .
do
  let n u64 stdio.write 1 "hello\n" . .   rem no capability was received
  return narrow u8 n. . .
end .
```

Diagnostic: `E-EFFECT-NO-CAP`

In plain words

The only difference between the two examples is one argument. The first received the output capability and the second did not. **There is no hidden standard output** — no function can put bytes outside without a capability.

6a There is no special notation for passing a capability on. The received name is written like any other argument.

7 An op that did not receive a capability cannot call an op that requires it. There is nothing to pass on.

Example — capabilities go down along the chain

Source

```
module ex_cap_chain .

use stdio .

rem The capability is received as an argument — its name is `k`, its type `cap io`.
proc say input k cap io . input msg slice u8 . output usize . effects io .
do
  return stdio.write k. 1 msg. . .
end .

rem The caller **just passes on by name** the `k` it received.
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 The entry point receives the capability **from outside** — nobody can make one themselves.
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 .
```

> **In plain words**
>
> The chain looks like this — outside → `k` of `main` → `k` of `say_twice` → `k` of `say` → `stdio.write` → the builtin op `write_out`. An op that did not receive the capability cannot call `say`. **There is nothing to pass on.** That is, what guards the boundary is not grammar but **arguments**. There is no rule to memorise, and by looking at the call sites one can follow by eye where the right came from.

> **Note (informative)**
>
> The value of this rule lies in being able to answer *"can this program touch files"* **by looking only at the entry point**. No function deep inside can secretly open a file — without having received the capability it cannot open one, and if it did receive one, that chain is visible all the way to the entry point.

> **In plain words**
>
> It can be compared to a key. To enter a room there has to be a key, and a key exists only when someone hands it over. Nobody can make up a key when none was given.

7.3 What purity means

An op that is `effects none` is *pure*. A pure op satisfies the following.
a) It always gives the same result for the same input.
b) Changing the order in which it is called does not change the result of the program.
c) If its result is not used, it need not be called.
A pure op cannot call an op that is not pure.
If an `fn` writes to storage it received as `mut`, that write is visible to the caller, so it is not pure (`E-EFFECT-PURITY`). Whether it writes directly, through a local name that sees that storage (rebound with `subslice` · `view_array`), or by passing it to a `mut` input of a `proc`, it is the same — if the receiver writes, the caller's storage changes. Passing it to a `mut` input of an `fn` is not counted here, since that `fn` is

checked by the same rule. Writing to storage built inside the op is not visible from outside, so it does not break purity.

> **Rejected example — an `fn` hands the `mut` piece it received to a `proc` to write**

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 the caller's piece changes
end .
```

Diagnostic: `E-EFFECT-PURITY`

7.4 Where other languages are met

1 Lowent can exchange with C. That boundary is marked with `extern`.

2 At the place that crosses the boundary, the language's guarantees do not carry over by themselves. So at the boundary **contracts shall be written**, and the processor enforces those contracts at the boundary.

3 The names going out to the C side are decided by the author. The processor does not make up names — that name is a promise made to another language, so the one who promises shall write it.

4 The name on the C side is written with the clause `link "<name>" .`. This is the «name the author decides» that (3) speaks of, and the processor does not make it up from the op's name.

5 An `extern` with the clause `variadic .` takes, after its fixed arguments, **arguments whose number is not fixed**. Contracts do not reach such arguments, so the caller is responsible for that place.

6 C can also be made **to call Lowent**. The address of an `export extern` op is obtained with `unsafe_fn <op>` and passed as a value.

7 At that place **the direction of the boundary reverses.** Handing it over is `unsafe`, since this language cannot know when, how many times, or in which flow it will be called, but **the place where the call comes in** is this side's door, so that op's contract is enforced on the arguments. That is, inside the door this language is responsible, outside the door C is.

> **Caution — A boundary is a door, not a hole**
>
> If the place where another language is met were left as *"from here on there is no guarantee"*, from that moment the guarantees of the whole program would become meaningless. Lowent **sets up contracts** at that place and makes people write what is promised to them and what they promise.

7.5 Doing things atomically

1 Operations with the `atomic` effect **are not split** — another flow cannot see their middle. There are reading · writing · adding · bit operations · swapping (swap) · compare-and-swap.

1a The result of an atomic operation that gives a value is of the element type of that slot. A read gives the slot's value, and add · subtract · bit operations · swap give the value **before the change**. Compare-and-swap (`atomic_cas`) gives 1 if it swapped and 0 otherwise, in that type.

1b Each machine has fixed widths for atomic operations done without a lock — the width for reads · writes and the width for read-modify-writes (add · subtract · bit operations · swap · compare-and-swap). If the element of the slot is wider than that, it is refused (`E-ATOMIC-TARGET`). In this

processor `cortex_m` is 4 bytes for both, `cortex_m0` is 4 bytes for reads · writes and has no read-modify-writes, and other machines go up to 8 bytes. The processor shows those numbers per machine.

2 `atomic_fence` computes no value and fixes **only order** — it makes what came before it visible before what comes after.

3 Atomic operations shall write their effect in the contract. Using them without writing it does not conform.

4 `order <name>` can be attached after an atomic operation to fix **what is visible to other flows, and when**. The names are the following five and no more; anything else is refused (`E-ATOMIC-ORDER`).

Table 51 — Memory ordering

Name	Meaning
`relaxed`	atomicity only, **no order**. Only that the value does not tear
`acquire`	what comes **after** this read does not leak forward
`release`	what comes **before** this write does not leak backward
`acq_rel`	both. Used for operations that read and write
`seq_cst`	every flow sees **one and the same order**. The strongest

5 If no order is written, it is taken as `seq_cst`. The strongest is the default because it is the easiest to reason about — weakening it is something done **by someone who writes it knowing its value**.

6 The orders that can be used differ by operation. **Combinations without meaning are refused** (`E-ATOMIC-ORDER`).

Table 52 — The orders each operation can use

Operation	Orders that can be used
read (`atomic_load`)	`relaxed` · `acquire` · `seq_cst` — a read cannot be `release`
write (`atomic_store`)	`relaxed` · `release` · `seq_cst` — a write cannot be `acquire`
fence (`atomic_fence`)	`acquire` · `release` · `acq_rel` · `seq_cst` — a fence with no order at all is not a fence, so `relaxed` cannot be used

> **In plain words**
>
> Why block combinations too? `atomic_load … order release` **makes sense as a sentence but has no meaning** — what would a read release? C left it as undefined behaviour, and undefined behaviour is silent. This language leaves nothing silent.

> **Note (informative)**
>
> Atomicity **is not free.** Using atomic operations on a value only one flow touches only makes that value slower. So this language does not make it the default but **has it chosen by name** — so that the cost is written in the signature.

7.6 Where devices are talked to

1 Reading and writing device registers is not an ordinary memory access. It is written with `read_volatile` and `write_volatile`, and the processor **shall not merge, remove or reorder** those accesses.

2 These accesses require the `device` effect and the matching capability (§7.2).

3 Reading or writing that place **like an ordinary field** is refused (`E-MMIO-PLAIN`) — that is, reading with `field <aggregate> <register>` or writing with `set field <aggregate>. <register> .`

`<value>` .. An ordinary field access is an operation the processor **may merge or remove**, so a device access written in that spelling **cannot keep** what (1) requires. If there are two ways of writing the same meaning and only one keeps that promise, the other is not a way but a trap.

> **Rejected example — touching a device register like an ordinary field**
>
> **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 an ordinary field write — an operation the processor may
> remove
>   return 0 .
> end .
> ```
>
> Diagnostic: `E-MMIO-PLAIN`

> **Note (informative)**
>
> The purpose of an ordinary read is *"knowing the value"*, so the processor may remove a second read. For a device register **the act of reading is itself the work** — reading sometimes changes the state. So it must not be removed, and the language has to know that.

7.6.1 Register blocks — `mmio`

1 A device's registers are written as an aggregate, and `mmio <start address> .` is attached to that aggregate to declare it **a map of the device**. The start address shall be an integer (`E-MMIO-BASE`) — a map has a start.

2 Opening that place without a map is refused (`E-MMIO-NOBASE`).

3 The address of a register block **comes from a capability.** Only an op holding `cap mmio` reaches that place (§7.2), and the start address is read as an address within the range that capability permits. On a machine whose pointers are capabilities (§5.3 (4a)), the block's pointer is **derived** from the capability received by the entry point — no pointer is made from an integer address alone. On a machine without such capabilities, the capability is only a mark, and the meaning is the same.

Table 53 — Places where a register block is refused

Diagnostic	What went wrong
`E-MMIO-BASE`	the start address is not an integer
`E-MMIO-NOBASE`	trying to open at its own address an aggregate that did not declare `mmio`
`E-MMIO-FIELD`	a register is not of a shape the machine can reach at once — its width shall be $1 \cdot 2 \cdot 4 \cdot 8$ and it shall be in its own place

`E-MMIO-BYVALUE`	trying to receive a register block **by value**. Receiving by value is copying, and **a device cannot be copied**
`E-MMIO-PERM`	trying to write a read-only (`ro`) register
`E-MMIO-PLAIN`	reading or writing a register like an ordinary field — it shall be written with `read_volatile` · `write_volatile`
`E-MMIO-NOCAP` · `E-MMIO-NOEFFECT`	the capability (`cap mmio`) or the effect (`device`) is missing
`E-MMIO-NOHOST`	trying to open an absolute address on a machine with an operating system — that address is not a device

> **In plain words**
>
> That a device cannot be received by value is the place in this clause people run into most often. Passing an aggregate by value is **copying**, and writing to the copy does not reach the device. A program written that way does nothing while saying nothing — so it is stopped at translation.

7.7 Where interrupts are received

1 Attaching a `vector <number> .` clause to an op makes that op **an interrupt handler**. To write its urgency as well, attach `priority <number> .`. No new word is introduced — clauses do that work.

2 An interrupt handler **is called by the machine.** So it shall keep the following four.

Table 54 — What an interrupt handler shall keep

To keep	If violated	Why
nobody calls it	`E-ISR-CALLED`	if our code calls it, the interrupt runs in a wrong state, **on the wrong stack and at the wrong priority**
no parameters	`E-ISR-PARAMS`	the machine passes no arguments
returns nothing	`E-ISR-OUTPUT`	there is nobody to return to — the machine woke it, and the machine resumes what was interrupted
writes `effects device`	`E-ISR-EFFECT`	it is a place that exists because of a device

3 Shared state passes between interrupt handlers and ordinary code by **a discipline of priorities and queues**. That discipline is not fixed by this document; it is the work of libraries (§9.5).

7.2.1 Where capabilities mismatch

1 Capabilities are judged **by kind**. Receiving a capability of another kind does not open that door (`E-CAP-KIND`). Not having one but **what it is** makes the authority.

1a Core ops and builtin ops that require a capability (input/output · terminals · clocks · random numbers · files · the network · the environment · arguments) **write that capability as their first operand.** Not writing it does not conform (`E-CAP-MISSING`) — merely having received the capability does not let it be used, and there is no way to reach a capability without a name.

1b The name written in that place is **the name of a capability this op received as an input**. A capability is not moved around like a value but handed from head line to head line (§7.2). So a line that **binds a capability to a local name** (`let` · `var` written with `cap <kind>`) does not conform by itself (`E-CAP-LOCAL`) — whether or not that name is used.

1c The same holds at **the place that calls an op** taking a capability as an input. The name written in that op's capability position shall be the name of a capability the calling op received as an input (`E-CAP-LOCAL`), the kind shall be the same (`E-CAP-KIND`), and an op whose first input is a capability

cannot be called without a capability (E-CAP-MISSING). Builtin ops that require capabilities are wrapped and provided by the standard library (§9.3 (1a)), so the place where an application program meets this rule is usually the line that calls that wrapper.

1d An op **does not give out** capabilities. An op that writes `cap <kind>` in its `output` clause — as it is, or inside another type such as `option` — does not conform (E-CAP-OUTPUT). An op that returns a capability becomes a second source invisible in the caller's head line. Spawning an actor with capability fields is specified separately by §8.6 (E-CAP-FORGE).

Rejected example — binding a received capability to a local name

Source

```
module ex_cap_local .

proc main input out cap io . output u8 . effects none .
do
  let k cap io out. .   rem this line is refused even if it is not used
  return 0 .
end .
```

Diagnostic: E-CAP-LOCAL

Rejected example — an op that gives out a capability

Source

```
module ex_cap_output .

fn screen input out cap io . output cap io .
do
  return out. .
end .
```

Diagnostic: E-CAP-OUTPUT

2 Requiring a capability the machine being built for cannot give is refused (E-CAP-NOHOST).

3 An op that writes the `alloc` effect and **does not receive a capability to obtain it** is refused (E-ALLOC-NOCAP). It wrote that it would use the heap but not where that place is received from. The capability is a `cap allocator` input, a region parameter, or a region block of a kind other than `heap`.

3a The `heap` effect is the same: written without a `cap heap` input or a `region <name> heap` block, it is refused (E-HEAP-NOCAP). `cap allocator` is the capability of a fixed window and does not authorise growing.

3b On a machine without an operating system (`machine.no_heap`), a growing root cannot be requested **from any place** — the `heap` effect, a `cap heap` input and a `region <name> heap` block are all refused (E-HEAP-NOHOST). On the same machine, `alloc` carving a fixed window can be used.

4 An op marked `unsafe` that does not write the `unsafe` effect is refused (E-UNSAFE-UNUSED) — the mark has meaning only when it reaches the caller, and what carries it there is the effects line. Conversely, doing such work without the effect is refused (E-UNSAFE-UNDECLARED).

4a `unsafe` **spreads all the way** along calls. There is only one place where it stops: an op that writes the clause `absorbs machine <name> .`. That op can call `unsafe` things in its body and **does not carry it in its own effects line**. And `<name>` is a `cap machine` inside that body — that is where that capability is born. No other kind can be absorbed (E-ABSORB-SCOPE): making up a capability that reaches the world would create an authority the caller cannot see.

4b An absorbing op shall have **all** of the following; if even one is missing, it is refused.

Table 55 — What absorbing requires

To have	Diagnostic	Why
an effects line of `none`	`E-ABSORB-IMPURE`	absorbing stops only `unsafe` — the caller has to know about other effects
`reference <op> .`	`E-ABSORB-NOREF`	without **a pure version** that gives the same answer, there is no way to tell «correct» from «consistently wrong»
one or more `requires`	`E-ABSORB-NOCONTRACT`	the inside cannot be seen but **the door can** — length · alignment · range are checked at the door
`why "…" .`	`E-ABSORB-NOWHY`	a promise whose assumptions are not written can be confirmed by nobody
permission in the manifest	`E-ABSORB-PLACE`	**who** may vouch is decided not by the source that wants that right but by the project's identity file

4b′ The manifest writes by name the modules permitted to absorb. A module not written there that absorbs is refused, and if there is no manifest at all, **nobody** can absorb — the default is no permission.

4c The processor **shall be able to say** where absorbing happened — which op absorbed what, and on what ground, one line each. A zone it cannot see is not hidden but left **visible**.

4d Absorbing is **a choice**, not a duty. An op that does not absorb passes `unsafe` on as now.

> **In plain words**
>
> Spreading is right — if the caller does not know what the processor cannot see, the effects line becomes a lie. But without a place where it stops, that door becomes **a door nobody can use** (measured 2026-09-23: uses of `asm` in the standard library, 0). So a place to stop is opened **narrowly and checkably**. The tool lists those places, and the ledger holds the signature and the hash of the body — what people agree to keep is either counted or disappears.

> **Rejected example — saying it will use the heap without receiving that capability**
>
> **Source**
>
> ```
> module ex_alloc_nocap .
>
> proc f output u8 . effects alloc .    rem there is no `input k cap allocator .`
> do
>   return 1 .
> end .
> ```
>
> Diagnostic: `E-ALLOC-NOCAP`

7.2.2 Where things come in and where they go out

1 The op at which a program starts receives **only capabilities** (`E-ENTRY-PARAMS`). It receives no data — there is nobody to hand it over.

2 The capabilities the starting place can receive are **a closed row** — `args` · `env` · `io` · `allocator` · `heap` · `file_system` · `net` · `process` · `tty` · `clock` · `random` · `atomic`. (`process` — the right to start other programs and watch them.) If the starting place requires any other capability, it is refused (`E-ENTRY-CAP`) — writing a right nobody can give is decoration, and **decoration is a lie.**

> **In plain words**
>
> Why only seven? The capabilities of the starting place shall be **what the executing side can actually hand over**. Capabilities not on the list (for example atomics · machine instructions · devices) **cannot be made up by the program itself**, nor are they something the executing side can hand over — such capabilities shall be made at a place entitled to give them and flow as arguments. Devices (`mmio`) and machine instructions (`machine`) are such, and so is the door into C (`c`).

3 Names of other modules can be used only if that side `exported` them (`E-VISIBILITY`). Writing the module in front of a name does not create an authority that was not there.

> **In plain words**
>
> The one thing these clauses repeat — **capabilities are handed over, not picked up.** The heap, the door into C, machine instructions, devices and the starting place are all the same. A power whose source is not written does not exist in this language.

8 Memory and ownership

1 This clause specifies **where values live and when they disappear**. Lowent has no garbage collector, and none of the mistakes that come from releasing by hand — in place of both there are translation-time rules.

8.1 Where values live

1 **A stored value** lives in some *region*. The region is fixed together with the type and is written in the source.

1a Not every value has a region. A small value that exists only during a computation may have no place to be stored, and such a value has no region. A region says **where a value is stored**, not how a value exists.

2 The regions of stored values are of three kinds.

Table 56 — Where values live

Kind	Explanation
local	born inside an op and gone when the op ends. The most common
static	exists for as long as the program lives
obtained	obtained from a root. Who gives it back is decided by ownership (§8.4)

2a There are **two** roots to obtain from. They are carved separately and rewound separately.

Table 57 — The two roots

Root	Carved with	Effect	Property
fixed window	`cap allocator` · regions other than `heap`	`alloc`	**does not grow.** When used up it gives `none`. On a machine without an operating system it lies between two bounds the linker gives — the size is not fixed into the executable, and that place is written when the device boots
heap	`cap heap` · `region <name> heap`	`heap`	**grows.** More is fetched during execution, and bytes already given are not moved. Exists only on machines with an operating system

3 Which region a value lives in **is written in the source.** The processor does not move it secretly.

3a Apart from what is obtained from roots, the values that `some` · `ok` · `spawn actor` make live in a **finite pool** of the processor (the place and lifetime of element lists `lit array` · `lit slice` are specified by §6.2.6 (7), and the place of struct value `lits` by §6.2.6 (7c-1) — they live in the op's frame, and when they go into the load of `some` · `ok` · a container · a channel · a letter · a task they are copied as values and live in the pool). This pool is the op's implicit ⟦frame⟧ — it has no effect, writes no `alloc` · `heap`, and needs no capability. The pool is rewound on every iteration of a loop, and if the values alive at the same time exceed the size of the pool, it **stops** on the spot (not a value but a halt). The size is decided by the machine (small on machines without an operating system), and the person building can adjust it. This size is not normative but a fact the processor states.

3a-1 For each entry op, the processor counts at translation the **upper bound** of values the deepest call path can leave in the pool — it adds up the number of places on that path that make values (a place inside a loop counts one iteration's worth · struct values living in frames are not counted). For paths that cannot be counted (recursion · piling up in fields · loops that cannot be rewound), it states the reason instead of a number. If the counted bound is larger than that machine's pool, the program **does not conform** (`E-POOL-BOUND`). This count is shown by `--stack-report` (the same place as the frame count of §6.2.6 (7c)).

3b A root's cursor is not shared between flows of execution. So an op spawned as a task cannot obtain from a root (`E-ALLOC-TASK`).

3c One allocator being **placed in two or more flows at the same time** at one place does not conform (`E-ALLOC-SHARED`) — handing it to two tasks, or handing it to a task while using it alongside. If its `reserve` moves the cursor atomically (the `atomic` effect), it is not so. Giving each task **its own allocator** conforms — there is one cursor each, so there is nothing to compete over.

> **In plain words**
>
> Until 2026-09-16, (3c) blocked «handing an allocator to a task» itself. So doing exactly what the diagnostic recommended — «give each task its own allocator» — was refused with the same diagnostic (defect note 75). **If the way a diagnostic recommends is blocked, that diagnostic closes a way rather than opening one.**

> **In plain words**
>
> What are commonly called «the stack» and «the heap» correspond to local and obtained. The names differ because in this language they are not machine structures but **properties of values**.

8.2 Places that receive regions

1 Regions **are handed over as arguments**. It is written `input <name> region <type> . .`, and the type after it is the name by which that region is called.

2 An op that obtains places from a region has the `alloc` effect (§7.1). That is, **there is no op that allocates secretly** — an op that uses places has that written in its contract.

3 What is obtained from a region cannot outlive that region. When the region ends, what was obtained from it ends too.

3a A region is not given back piece by piece. It is collected **all at once** when the region ends.

> **Example — receiving a region and obtaining places**
>
> **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 .
> ```

> **In plain words**
>
> Note that `effects alloc` is written. Obtaining places is **an effect**, so a pure op (`fn`, `effects none`) cannot use regions. It means that where memory comes from is written in the signature.

4 The statement that opens a region writes its name together with its **kind**: `region <name> <kind> do … end .`. The kinds are a closed eight — `stack` · `frame` · `arena` · `static` · `heap` · `mmap` · `disk` · `device`. Any other word does not conform (`E-REGION-KIND`).

4a **Carrying out** of a region a value obtained inside it does not conform (`E-REGION-ESCAPE`). When the region closes, that value no longer exists, so a name left outside points at what does not exist. The places of carrying out are `return`, assigning to a name outside the region, and assigning to **a field or element** of a name outside the region (`set field h. store . b. .`). Values that do not hold the region's bytes — those bound to scalar types like integers · truth values, and expressions that give scalars like `len` · `idx` — are not carried out. Whether something holds the region's bytes is decided **by following the flow**: the result of an op call holds only what belongs to the inputs that flow into the result in that op's body. If the result is an aggregate built with `lit`, each field is decided separately — taking out and carrying away a field that holds the region's bytes does not conform, and fields that do not hold them do not matter. If the flow cannot be traced, as when the body puts the result into a local and then returns it, it is regarded as coming from all the inputs.

> **Rejected example — putting a slice of the region into a field of an outer aggregate**

Source

```
module ex_region_field .

struct holder do store mut slice u8 . end .

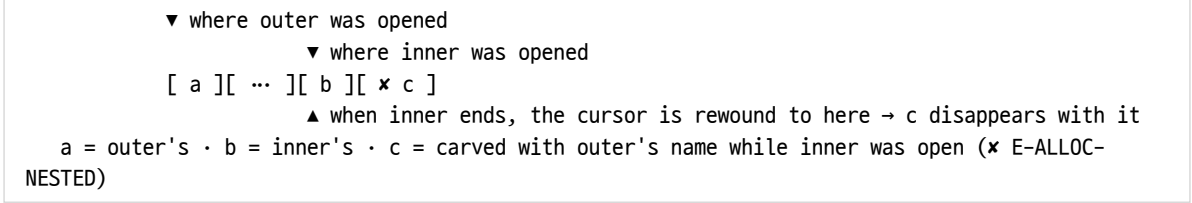
proc f output u64 . effects alloc . do
  var h lit holder do store subslice "abcd" 0 0 . . end . .
  region r arena do
    let g option mut slice u8 alloc_bytes r. capacity 16 . .
    if is_some g. . do
      let b mut slice u8 some_value g. . .
      set field h. store . b. .
    end .
  end .
  return len field h. store . . .
end .
```

Diagnostic: `E-REGION-ESCAPE`

4b Opening a region with a name that is not a region parameter of that op does not conform either (`E-REGION-UNDEF`).

4c **Carving with the name of an outer source** while a region of the same root (§8.1 (2a)) is open inside does not conform (`E-ALLOC-NESTED`). This holds for the name of an outer region block, for a region parameter, and for a capability of the same root (`cap allocator` for the fixed window · `cap heap` for the heap). Each root has one carving place, so when the inner region ends, those bytes are rewound too. A region of another root being open inside does not matter.

> **Diagram — one cursor per root — when the inner region ends, the cursor is rewound**



Rejected example — carving from the outer region while the inner region is open

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 .
```

Diagnostic: `E-ALLOC-NESTED`

4d Sending bytes obtained inside a region to **an actor born outside the region** does not conform (`E-ALLOC-OUTLIVES`). The actor outlives the region, and whether it keeps the slice it received is not visible at translation. Make the actor inside the region, or hand over memory that outlives the region.

4e On **every path** that leaves a region block — reaching the end, `return`, `break` · `continue` of a loop enclosing the region — that region is rewound. If the name of a region block is passed as an argument, the receiver takes it as a region parameter and carves from the same root.

5 The kind decides **which root it carves from** (§8.1 (2a)): `heap` carves from the heap, and the other seven carve from and rewind the fixed window. Regions of the two roots can be opened overlapping within one op, and rewinding the inner region leaves outer values of the other root in place. The processor of this edition **does not treat** the seven other than `heap` **differently from one another** — those seven are words that tell the reader **where the storage comes from**, and having the processor distinguish them is the work of a later edition.

Note (informative)

Closing the vocabulary and separating behaviour are different things. If any word could be used, that word would say nothing — because it was closed at eight, someone who reads `region t arena` knows *"it is bump allocation and is given back all at once"*. The distinction of behaviour goes as far as **the two roots** (fixed window · heap), and there is no distinction yet among the other seven. That fact is written down here.

8.3 Moving and copying

1 When a value is put into another name, if that value has ownership it is **moved** (the original name can no longer be used). If it is a value without ownership, it is copied.

2 Using the original name after it has been moved is refused at translation.

8.3.1 Where a value is put — temporaries and destinations

1 Sometimes an expression that makes a value has **a place where it will be put** (a *destination*). The initial value of `let`, the value of `return`, argument positions and the fields of aggregates are such.

2 If all of the following three hold, that value **is made directly in the destination** — neither an intermediate temporary nor moving it happens.

Table 58 — The three conditions for being made directly

Condition	Meaning

the destination **is empty**	`let` initialisation · a field not yet filled · a place emptied by a move
the value **does not fail midway**	there is no path that leaves before reaching the destination
the destination **belongs to that value alone**	two values do not compete for the same place

3 If the conditions do not hold, a *temporary* is made. A temporary is put in the smallest enclosing scope, and when that scope ends the temporaries disappear **in the reverse order of their making**.

4 A reference pointing at a temporary cannot go outside that scope (§8.4.1).

5 There is no separate operation that moves a value. Moving is **changing places**, not making a value.

> **Note (informative)**
>
> Why write «made directly» as a norm? Left as *"do it if possible"*, the same source would copy on some processors and not on others — then **the cost would not be visible in the source**. The three conditions above are things a reader can count from the source alone, so when they hold one **may believe there is no copy**.

> **Note (informative)**
>
> C++ solves this place with «value categories» and «move constructors». Lowent does not bring in that vocabulary — moving is **the business of places**, not of types. So there is no special declaration for moving, nor any rule that calls one.

8.4 References

1 A *reference* points at a value somewhere else. There are two kinds.

> **shared reference** (ref)
>
> A read reference. Several can hold one at the same time. A value cannot be changed through it.

> **exclusive reference** (mut_ref)
>
> A write reference. For one value there can be **only one at a time**, and meanwhile there can be no read references either.

2 Violating this rule is refused at translation.

3 A reference is not a value. Using a name that received a reference as it is in arithmetic or comparison does not conform (`E-TYPE-REFVAL`) — to use the value pointed at, take it out **in writing** with `deref`.

4 This edition has no notation for borrowing a field of a struct separately. Writing a field borrow does not conform (`E-BORROW-FIELD`) — borrowing takes the whole value as its unit (§8.12).

> **Rejected example — using a reference like a value in arithmetic**
>
> **Source**
>
> ```
> module ex_ref_value .
>
> fn twice input p ref u64 . output u64 .
> do
> ```

```
  return add p. p. . .    rem the value pointed at is taken out with `deref p`
end .
```

Diagnostic: `E-TYPE-REFVAL`

8.4.1 A reference cannot outlive what it points at

1 A reference pointing at a local value remaining after that value has disappeared is called an *escape*. Lowent **refuses escapes at translation time**.

2 That is, a reference pointing at a value made inside an op cannot be returned outside that op.

Example — several can hold read references together

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 .
```

Rejected example — a reference pointing at a local value cannot be returned

Source

```
module ex_escape .

export fn leak output ref u32 .
do
  let here u32 42 .
  return ref here . .     rem `here` disappears when this op ends
end .
```

Diagnostic: `E-ESCAPE: reference to a local escapes the op (dangling)`

In plain words

If the code above were translated, the returned reference would point at **a value that has already disappeared**. If that reference were read, nobody knows what value would come out. So this language does not accept that code at all — it is known **at translation**, not by running it and finding out.

Caution — There are no null references

A reference always points at a live value. If *"pointing at nothing"* has to be expressed, use `option` (§6.2.8) — then the processor enforces a check before taking it out.

Note (informative)

The rule that separates many readers from one writer prevents two things at once. One is **another flow changing it at the same time**, and the other is **the value changing while being read**. Both are kinds of mistake that cannot be known by reading the source, so they are removed by rule.

8.5 Ownership and destroying

1 *ownership* says who is responsible for destroying a value. `owned t` is a value that has ownership.

2 A value with ownership shall be destroyed **exactly once**. Destroying it twice, and discarding it without destroying it, are both refused at translation time.

> **Diagram — ownership moves around — the responsibility to destroy is always in one place**

```
 var h be owned buffer v .      h ──► [ buffer ]            responsibility: here
 consume h .                    h (empty name)  ──move──► h of consume ──► [ buffer ]
responsibility: consume
 drop h .   (inside consume)                                  [ buffer ] gone
responsibility: ended
 using h again                  E-OWN-MOVED                 (use after move)
```

3 Destroying a value is of **two kinds**, and this language treats the two differently.

> **release** (해제)
>
> Work that **always succeeds and does not wait**, like giving memory back. It has no failure, so it **may happen silently** where the value's lifetime ends. There is no failure to swallow.

> **completion** (완결)
>
> Work that **can fail and can wait**, like closing a file · flushing a buffer. If it happened silently, **there would be no place to hand that failure to.** So the author shall always call it in writing.

4 Whether a value requires completion is **decided by the program's own declarations** — if there is an op that takes that type by value and returns a `result`, that is the declaration *"ending this can fail"*.

5 Discarding without calling it a value that needs completion does not conform. The processor issues a diagnostic (`E-OWN-INCOMPLETE`).

6 Using a moved value again does not conform either (`E-OWN-MOVED`).

7 When binding a value that needs completion, `owned` **shall be written in that place** (`E-OWN-BARE`). Without that word nothing requires ending, and what the type says and what the binding says part.

8 At a place where branches split and meet again, the state of ownership **shall be the same on every path** (`E-OWN-JOIN`). If it is destroyed on one path and kept alive on another, **nobody can say** whether that value is alive after they meet.

8a A path that leaves a loop is a path too. A path that leaves a loop with `break` meets the ordinary end after that loop — if the value was moved on only one side, it is refused (`E-OWN-JOIN`). A `break` with a number meets after the loop that many levels outward. A `while` whose condition is `true` ends only with `break`, so there is no path to meet — the state of that path is the state after. Going to the next iteration with `continue` after moving a value is refused, since the next iteration meets the moved value again (`E-OWN-MOVED`).

9 After moving one `owned` field of an aggregate, **moving the whole aggregate again** does not conform (`E-OWN-PARTIAL`). The receiver believes it received a whole aggregate, but one field inside it already belongs to someone else.

> **In plain words**
>
> These three are all different faces of the same question — that *"who is responsible for destroying this value now"* shall **always have one answer**. If the answer differs by branch (8), or only part has moved (9), or the question was never asked (7), the end of that value becomes a place nobody is responsible for.

Example — release — it cannot fail, so `drop` is enough

Source

```
module ex_own .

type buffer u8 .

fn sink input h owned buffer . output u8 .
do
  drop h. .
  return 0 .
end .
```

Rejected example — it cannot be destroyed twice

Source

```
module ex_own_bad .

type buffer u8 .

fn twice input h owned buffer . output u8 .
do
  drop h. .
  drop h. .     rem destroys again what is already gone
  return 0 .
end .
```

Diagnostic: `E-OWN-MOVED`

In plain words

It matters that there is only one criterion separating the two kinds — **can ending it fail.** If it cannot fail, no information is lost when the language cleans up by itself. If it can fail, someone has to receive that failure, and the only one who can is the author.

Note (informative)

It is also worth noticing that no new word was made. Completion is **an ordinary op** — a function that takes an owned value and returns a `result`. That shape itself becomes the declaration, so there was no reason to add a word to the language.

In plain words

Catching «forgot to destroy» at translation time is the core of this clause. It does not expect the operating system to clean up when the program ends, so the same code holds even in an environment without an operating system (§5.1).

self-owning node (스스로를 품는 마디)

A struct one of whose fields is `option (owned <its own type>)`. That field is not a value but **one address slot** of another node built in allocator bytes, so the size is fixed, and nodes are linked to build trees.

10 A *self-owning node* is a struct with a field `<name> option (owned <itself>) .` (§6.2.7 (3b)). Nodes are built only in bytes an allocator gave — the binding's `use` chooses that allocator: `let n use al. option (owned node) lit node do … end . .`. If there is not enough room, the value is `none`. Nodes are

treated like **values that need completion** — discarding an `owned node` · `option (owned node)` without moving or returning it (end of block · an early `return`), or overwriting a value still held, does not conform (`E-OWN-INCOMPLETE`).

11 A node is given back only with `drop <name>. use <allocator>. ..` If that allocator gives back piece by piece (`freeing_allocator`), the bytes go back to it; otherwise the allocator keeps holding the bytes. If the name is an `option` and empty, it does nothing. `drop`ping a node without an allocator, or writing `use` on a value that is not a node, does not conform (`E-DROP-USE`).

12 Giving back a node whose child fields still hold something does not conform (`E-OWN-NODE-KIDS`) — move the fields out first. If that node is inside an `option` and not visible at translation, execution stops on the spot (`E-VM-NODE-KIDS`). Deep trees are torn down with loops, not recursion.

13 Taking a child out of a node borrowed with `ref` does not conform, and a child taken out of a node borrowed with `mut_ref` shall be put back before the op ends (`E-OWN-NODE-REF`) — the lender gets back a node with a hole. Only reading (`is_some field t. left .`) is not taking out. Moving as a whole a node whose child was taken out does not conform, as in (9) (`E-OWN-PARTIAL`).

Example — a borrowed node is only read

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 .
```

Rejected example — a node is given back to its allocator

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 .
```

Diagnostic: `E-DROP-USE`

In plain words

Nodes do not hold their allocator because it would be waste — the head of a tree holds one allocator. So a node does not know «where it came from», and the one who writes where to give it back is the author. If the node type is not exported from its module, the mistake of giving one tree's nodes back to another allocator is confined to that module (the standard library `tree`).

8.6 How far memory safety is guaranteed

1 This document **writes the strength** of its guarantees **separately.** Otherwise it would make promises it could not keep.

Table 59 — Strengths of guarantee

Strength	Meaning
proved	there is a machine-checked proof
static	prevented by translation-time rules. There is no proof of the rules themselves yet
dynamic	checked during execution. Violations trap

2 The exclusivity rule of references, the ban on escapes, and destroying owned values once are **static**.

3 Slice bounds and contracts are **dynamic**, but checks disappear when the processor proves them (§6.4.6).

> **Caution — The word «safe» means something only when it says what is safe and how**
>
> When a language claims *"memory safety"*, it matters whether that is prevented at translation time or checked during execution, and whether it is proved or only a rule. This document writes the strengths separately **so as not to make promises it cannot keep**.

8.7 Reading bytes again — `bit_cast` and `view`

1 **Reading the same bytes as another type** is divided into two kinds in this language. They are divided because if that work were put off wholesale as «unsafe», nobody could say how far the harm spreads.

Table 60 — The two kinds of reading bytes again

Kind	op	What it is
moving a value's bits as they are	`bit_cast`	between two scalars of the same width. The bit pattern is left as it is and **only the way of reading** changes
laying a shape over bytes	`view` · `try_view` · `view_array`	**reads** a piece of bytes with the layout of some shape. It does not copy

8.7.1 `bit_cast` — the bits as they are, only the way of reading

2 The target type of `bit_cast` shall be **a `plain` scalar**. `plain` means a type in which **every bit pattern is a useful value and there is no padding**.

3 `bool` and *enum* are **not** `plain` — because they have useless bit patterns (*trap representations*). So these two are outside this kind and are refused (`E-TYPE-BITCAST`).

3a Source and target shall have **the same width**. If the widths differ it is refused at translation (`E-TYPE-BITCAST`) — nothing can say where the extra or missing bits come from. Change the width first with `widen` · `narrow`.

> **Caution —**
>
> «Is it `plain`» shall be asked **directly**. A way that asks indirectly, treating what is not in the width table as not `plain`, silently flips the day the width table is filled. A verdict that held by coincidence disappears without a word when that coincidence disappears.

> **Rejected example — `bool` has trap representations, so it is not `plain`**

Source

```
module ex_bitcast .

fn f input a u8 . output bool .
do
  return bit_cast bool a. . .     rem not every bit pattern is true or false
end .
```

Diagnostic: `E-TYPE-BITCAST`

8.7.2 `view` — laying a layout on

4 `view` lays the layout of a shape over a piece of bytes. **It does not copy** — it reads the bytes in that place as they are.

5 Two things shall fit for it to be laid on.

Table 61 — The conditions for `view` to stand

Condition	If violated	Why
the piece is as long as the layout	`E-VM-VIEW`	reading missing bytes is exactly what this language prevents
the start address keeps `align`	`E-VM-ALIGN`	`align n` is **a contract**, and this boundary is exactly where the contract is checked

6 `try_view` does the same, but if it cannot stand it gives **absence instead of a lie** (⟦option⟧). That is, the failure comes back as a value.

7 `view_array` lays **several** of the same shape on. The conditions are the same.

7a **Row views** — `view_array array m t b.` lays one row (`m × ¦t¦` bytes) on as one element. The result is `slice array m t` (the same type as the sequence of rows of §6.2 (9a)), and `idx g. r.` is a `slice t` that sees row r. The conditions are the same as (7) — if the length of `b` is not a multiple of the row bytes it stops (`E-VM-VIEW`), `t` shall be `plain` (`bool` is `E-TYPE-BITCAST`), and `m` is a literal of 1 or more. Whether it can be written follows `b`, and its source is `b` (§8 (8a)) — it does not copy, so the rules of invalidation · escape stand as those of `b`.

> **In plain words**
>
> If alignment is treated as «a matter of speed», places that break it pass silently and one day collapse only on some machine. This language calls it **a contract** and checks it at the boundary — it refuses misaligned addresses for **exactly the same reason** it refuses short pieces. Both are «reading what is not there as if it were».

8.8 Changeability flows only as written

1 Pieces and references are **read** by default. To change them, `mut` shall be written in that place.

Table 62 — Places where changeability does not match

Diagnostic	What went wrong

E-TYPE-MUT	writing to an element of a piece not written mut
E-TYPE-REF	writing through a shared read reference — to change it, it shall be mut_ref
E-TYPE-ARGMUT	passing a read-only value to a place that takes mut · owned · mut_ref
E-TYPE-RETMUT	writing that it gives something changeable and returning something read-only

2 That is, changeability narrows **in one direction only**. Passing something changeable as read is allowed, but passing something read-only to a changeable place is not.

> **In plain words**
>
> What this rule protects is **the reader's trust**. If I received a value as read, I shall be able to trust that it does not change while I am looking at it, and that trust stands only when nobody secretly obtains the right to write. So there is only one direction of narrowing.

8.9 Same name or not, types differ

1 Even with the same representation, **different names are different types** (E-TYPE-NOMINAL). Those separated with newtype, and different integer types of the same size, are such.

2 An undeclared type name is refused (E-TYPE-UNDEF).

3 Integers and floating-point numbers **do not mix by themselves** (E-TYPE-MIX). To cross, use the op written for it.

4 A type holding itself is refused (E-TYPE-CYCLE). Even when the processor cannot follow that recursion to the end, it **does not pass silently** but says so (E-TYPE-CYCLE-LIMIT).

> **Rejected example — integers and floating point do not mix by themselves**
>
> **Source**
>
> ```
> module ex_mix .
>
> fn f input a u32 . input b f64 . output f64 .
> do
>   return add a. b. . .     rem to cross, use the op written for it
> end .
> ```
>
> Diagnostic: E-TYPE-MIX

8.10 Storage the linker gives — storage reserved

1 A storage reserved . block opens static storage **that the linker sets aside for that type**. It is not obtained by the program during execution; its place is already fixed when it is built.

2 An op that opens this block shall have the following three.

Table 63 — What storage reserved shall have

To have	If missing	Why
a capability	E-RESERVE-NOCAP	a place the linker hands over is also **handed over**, not picked up
effects state	E-RESERVE-NOEFFECT	it is a place that remains while the program lives, so touching it is touching state
the machine can give it	E-RESERVE-NOHOST	on an operating system, the linker does not give that place that way

8.11 Parentheses do not cross blocks

1 An opened (shall be closed **within the same block**. If the block ends while it is unclosed, it is refused (E-PAREN-ESCAPE), and if it is never closed, it is refused (E-PAREN-UNCLOSED).

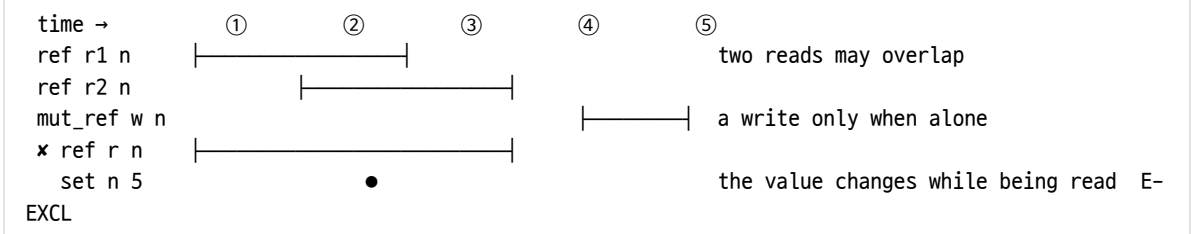
2 A) without a partner, and a) that looks for its partner across a block boundary, are refused (E-PAREN-STRAY).

> **In plain words**
>
> If parentheses could look for their partners across blocks, a source missing one parenthesis could be **read wholly with another meaning**. Then the processor would build a different program instead of reporting the mistake. If they cannot cross the boundary, an omission always shows as an omission.

> **Rejected example — the parenthesis is not closed before the block ends**
>
> **Source**
>
> ```
> module ex_paren .
>
> fn f output u8 .
> do
>   return (add 1 2 .
> end .
> ```
>
> Diagnostic: E-PAREN-ESCAPE

8.12 Exclusivity — many readers or one writer

1 The ways that reach a value at one moment are **many reads** or **one write**, and those two overlapping does not conform (E-EXCL). If they overlapped, nobody could say **when** the value the reader sees changes.

> **Diagram — borrows of one value n, in time order**
>
> 
>

2 What is borrowed **cannot outlive** the value it was borrowed from. If the lent value has been moved while the borrow remains, it does not conform (E-EXCL-MOVED).

3 A borrow **cannot go outside** the block that opened it (E-BORROW-ESCAPE).

4 Holding something borrowed from an actor and talking to that actor again does not conform (E-BORROW-EXCL) — while that actor changes its own state, we are looking at that state.

5 A name bound with `let` **cannot be changed** (E-IMMUTABLE). Bind a name to be changed with `var`.

6 One call passing the same storage to **two or more `mut` positions** does not conform (E-EXCL). There are two writers, so it violates (1) directly. Which write remains is decided not by the calling place but **by the order of the callee's statements**, and the caller's text has no way of knowing it.

6a Whether it is the same storage is decided not by name but **by place**: an alias of one name, overlapping pieces cut from the same storage, **the same field** of the same value, and **the whole** of a value **and its field** are the same storage. **Different fields** of one value (`field s. a .` and `field s. b .`) are different storage — passing a field to a `mut` position passes that field's place, so two different fields can be passed together to two write positions.

7 Passing the same storage together to **one** `mut` position and a reading position (an in-place operation) conforms only when the callee has **declared** the pair of those two inputs with the clause `inplace <write input> <read input> .` (§6.4.1 (3a)) and the two storages are **the same range** (`E-EXCL-INPLACE`). The same range means the whole of the same storage, or pieces cut from the same storage with the same start and the same end. If the ranges overlap out of step or one is part of the other (**partial overlap**), it does not conform even with the declaration — a body that reads and writes element by element is correct only on the same range, and the caller cannot know where in the storage the callee writes.

7a The `inplace` clause writes **two** names: the first is a `mut` input of that op, the second another input of that op. One clause is written per pair. A mismatched shape is `E-INPLACE-FORM`. Whether that declaration **is true** — whether the body runs correctly on the same range — shall be shown by one of two shapes of the body: ⓐ it does not write to the write input up to the last statement that reads the read input, or ⓑ within that last statement it writes only with one index V as `set idx <write>. V. .` and reads only with the same index as `idx <read>. V. .`, and, inside a loop, does not read after writing in the same iteration (in place, element by element). Passing the same pair to an op that declared it `inplace` is left to that op's declaration. If it is not shown by either shape, it is `E-INPLACE-UNPROVEN`. This check is conservative — it may refuse a correct body but never admits a wrong one, and a refused body is rewritten as ⓐ or ⓑ.

7b When whether two ranges overlap cannot be decided at translation (when a cut end is not a constant), the processor asks (6) · (7) **during execution**. On entering, an op looks, for each pair of slice · view parameters — at least one of them `mut` — at whether the two pieces overlap by even one byte, and stops if they do (`E-VM-EXCL`; natively a panic with the same sentence). It passes if that op allowed the write · read pair with `inplace` and the two pieces are the same range. An empty piece overlaps nothing. Overlap is decided by byte places, so two windows with different element widths overlap too if they see the same bytes. Calls already decided at translation (constant ranges) are refused with the diagnostics of (6) · (7), and this check does not change the answers of calls that do not overlap. This check looks only at pieces that are parameters — pieces inside struct fields are not compared. The processor may drop the comparison of one pair from an op's entry. But only when every way of reaching that op within the translation unit is a call site (not reached through function values · actors · tasks), and **every** such site already shows that the two pieces cannot overlap. The shapes that show it are four: ⓐ two pieces cut from the same storage are empty, or the end of one piece does not pass the start of the other — an end is a constant, or an expression that reads, for the same run-time value `v`, as `k + m·v` (`k` · `m` constants, $m \geq 0$) (from 0 up to before `k` and from `k` up to before `n` · from 0 up to before `n` and from `n` up to before `2n`). If that pair was allowed with `inplace`, it may be the same range. A piece seen as other elements with `view_array` is the same bytes. A piece cut again from a piece with a constant end (or that piece seen with `view_array`) lies **inside** that piece even if its end is unknown — `subslice` stops execution if the end is outside (`E-VM-BOUNDS`) — so if the two enclosing constant ranges split front and back, they do not overlap, ⓑ the two pieces are two inputs of the calling op (or pieces inside them), and the calling op has already compared that pair as not overlapping at its own entry (for a pair compared with `inplace`, only when both are passed as they are and the receiver also allows that pair with `inplace`), ⓒ one piece is a string literal (or a piece of one), or a name whose value is always a string literal — a writable place cannot point at read-only bytes (§6.2.6 (7b)), ⓓ one piece is a piece of a list the calling op built itself (a slot in the frame of §6.2.6 (7) ⓐ) or of bytes that op obtained with `alloc_bytes` (its `some_value`), and the other piece is a list of another declaration · the answer of another `alloc_bytes`, or an input of the calling op — each declaration and each obtaining has its own bytes (a root does not give bytes it gave again, and rewinding happens only when a region ends — §8.1 (2a) · §8.2 (4c)), so they do not overlap each other, nor bytes that existed before the call. Taking the same answer out into two names gives the same bytes. A name that binds a piece with `let` is seen as the value of that `let` — only when that

name and the names in the value are bound once in the op and not lent by reference (for an input, only when the body does not write it again). An entry from outside the translation unit (a name C calls) compares the dropped pairs at that entry. So a call that can overlap is compared and stopped whichever way it comes — dropping changes neither answers nor stops.

In plain words

Until 2026-11-04, (7b) was «not asked». So `wr2 (subslice s 0 k) (subslice s 0 k)` passed the check and the later write overwrote the earlier one (external report REQ-0036). Of three options — treating the same name and same-number ends as the same range at translation, writing it down as a limit and allowing it, or comparing during execution — the owner chose the third (2026-11-04). When comparing began, three calls in the corpus stopped: two that passed a whole buffer and a window cut inside it with `view_array` to writing places (the static check could not follow that alias), and one where the wrapper of an op that declared it takes the same range had not copied over that permission (`inplace`). All three were fixed to cut the storage without overlap or to write the permission.

In plain words

Until 2026-09-21, (7) was **the author's duty** for «ops that write only after they have finished reading», and the tool did not look (RFC-0115 §8-15 ©). So violating calls showed up not as diagnostics but as wrong answers. Now the permission **is written in the op's head** and can be asked at the one place of the caller (RFC-0116 D2 B1, owner decision ⓑ of 2026-09-22). Partial overlaps the old rule allowed are blocked by the new rule — the seven such calls in the corpus were fixed to cut the storage without overlap.

Rejected example — passing the same storage as write and read together to an op that did not declare it

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 .
```

Diagnostic: `E-EXCL-INPLACE`

Example — an in-place call of an op that allowed the same range

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 .
```

Rejected example — writing inplace on a body that reads after writing on the same range

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 .
```

Diagnostic: `E-INPLACE-UNPROVEN`

8 An op can write the clause `invalidates <input> .` (§6.4.1 (3a)) in its head to declare that it **makes views that came from that input's storage invalid** — giving a block back, moving to another place while growing, and rewinding are such. It writes **one** name, and it shall be an input of that op (`E-INVALIDATES-FORM`).

8a Within the body of one op, a view (a local or borrowed name whose type contains `slice`) carries as its **source** the storage that appeared in its initialiser — making a view from a view continues the source. Using that view after passing that storage to an op that declared `invalidates` does not conform (`E-VIEW-INVALIDATED`). Reassigning the name as a whole revives it with a new source. If it becomes invalid on **any one branch**, it is invalid after they join, and in a loop the invalidity continues to the head of the next iteration. **Values** such as handles and numbers are not views.

8b The sources of views **are lumped together** — the result of a call is regarded as coming from all its arguments. So it may refuse safe uses but never misses an invalidated view. Op boundaries are crossed like this:

- Even an op that does not write `invalidates` is regarded as an op that invalidates an input if, in its body, it passes that input to an op that declared `invalidates` (including ops so inferred)

— invalidation does not disappear by going through a wrapping op.

- If a view is put into a field of an aggregate (`set field h. f . <view> .`), that field holds the view. Reading and using that field after the field's source becomes invalid is

`E-VIEW-INVALIDATED`, and reassigning the field revives it.

8c Two `mut` views obtained by calling an op without side effects (`effects none`) twice **with the same arguments** are regarded as the same storage. Keeping both alive outside a `borrow` and using them violates (6) (`E-EXCL`) — taking out two write views with the same handle of a pool is such. If the arguments are not names or constants, whether they are the same is not asked.

Rejected example — using an old view after an op that moves while growing

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` invalidated the views of `b` — take the view again
end .
```

Diagnostic: `E-VIEW-INVALIDATED`

In plain words

What this rule protects is the same as §8.8 — **the reader's trust**. But where §8.8 decides «can it be written» by type, this one decides «who is here together right now». There are places where, even with the types right, things must not be together, and seeing them is the exclusivity rule.

Caution —

`mut ref slice` is not used (`E-MREF-SLICE`) — `mut slice` already lets its elements be changed, so the former **gives nothing the latter does not.** It is a second spelling that says the same thing.

Rejected example — a name bound with `let` cannot be changed

Source

```
module ex_immutable .

fn f output u8 .
do
  let a u8 1 .
  set a. 2 .            rem to change it, it shall be bound with `var`
  return a. .
end .
```

Diagnostic: `E-IMMUTABLE`

8.13 Allocators — capability · policy · state

allocator (얼로케이터)

A value that **hands out small places** from bytes received from a root or borrowed. In this language it is an actor that provides the trait `byte_allocator`.

1 What hands out places over a root (§8.1 (2a)) is called an *allocator*. An allocator is not a special device of the language but **an actor that provides the trait `byte_allocator`**. There are three things in that word.

Table 64 — The three that make up an allocator

Layer	When it exists	What it is
capability	only at translation	whether a root may be reached — `cap allocator` · `cap heap` (⟦§7.2⟧)
policy	at translation (the type)	by which rule it carves — **a type** that provides `byte_allocator`
state	during execution	the cursor · backing bytes — **a value** of that type

2 Capability fields (`root cap heap .` · `root cap allocator .`) can be put in an actor's state. Calling `alloc_bytes` with that field carves from that capability's root. A capability is not a run-time value, so that field has no size.

3 An actor with capability fields can be spawned only in **an op where a capability of the same kind is visible** — one that received that capability as an input, or an op of an actor that itself has a field of the same kind (`E-CAP-FORGE`). This is so that a way of reaching a root in one line cannot be made up in a place without the capability.

4 On a machine without an operating system, actors with a `cap heap` field and ops of actors that write the `heap` effect are refused (`E-HEAP-NOHOST`) — except that actors spawned nowhere in that unit are not asked about.

5 The standard library gives two default allocators in this shape: `allocs.fixed_bytes` (fixed window) · `allocs.heap_bytes` (heap). So default allocators and allocators the user built go into **the same place** — the receiving side (`input comptime a type . input al a . requires allocs.byte_allocator a. . .`) does not tell the two apart.

6 The `heap` effect means only «the root grows». **Giving places back piece by piece** is the work of policy, and such an allocator provides the trait `freeing_allocator` — in addition to the ops of `byte_allocator`, `release` takes a piece and answers with a truth value whether it took it back. So even on a machine without an operating system, an allocator that gives back piece by piece over a fixed window can be used.

7 When an allocator asks «is this the piece I gave last», it asks **by identity, not by length**: `grow` and `release` take the piece itself, and the core op `same_slice a. b. .` checks whether two slices start at the same place and have the same length. Recognising by length alone would grow or take back someone else's piece of the same length. `same_slice` does not expose addresses — the answer is a truth value.

> **Rejected example — spawning an actor with a capability field without the capability**

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 there is no `input h cap heap .`
```

```
  return send g. take 8 . .
end .
```

Diagnostic: E-CAP-FORGE

8.14 Choosing an allocator per object — the use clause of an op head and the use · keep of bindings

1 An op can write the clause use <name> <type> . in its head to declare **the allocator it carves from**. <type> is a type that provides byte_allocator, or a comptime type parameter with such a bound (requires allocs.byte_allocator a. . .). In the body, <name> is an ordinary name.

1a An op writes **only one** use clause (E-USING-DUP). The clause is two words, a name and a type. The allocator clause of a binding is written directly after the name as use <source> (gives back at the end of the block) or keep <source> (does not give back) (§6.5.1 (3c)) — a mismatched shape is E-USING-FORM.

1b The receiving side and the choosing side use **one word** — use <name> <type> . in an op head is «this op carves here», and use <source>. of a binding is «this object with this». The use of a head is told apart from importing (§6.10) by place: importing stands outside declarations, and this clause stands between the op's name and do. A head clause has no distinction between giving back and carrying out — keep stands only on bindings. The old spelling using <name> <type> . is refused (E-USING-OLD).

2 The use clause **is not an input.** The caller does not write that allocator positionally, and if <type> is a type parameter, it does not write that type argument either — it comes from the allocator's type.

3 The caller chooses the allocator at the binding: let <name> use <source> <type> <expr> .. The chosen source goes into the **head call** of that initialiser. If the head call is an op without a use clause, it does not conform (E-ALLOC-USING-UNUSED) — an object that already holds its own allocator does not take the caller's choice. If the initialiser is a lit, its bytes are received from the chosen source (§6.2.6 (7e)). A self-owning node (§8.5 (10)) is bound as owned, so it is not given back at the end of the block — the responsibility for giving it back is carried by that value, and the drop … use of §8.5 (11) gives it back.

4 If nothing is chosen, the default is decided in the following order. The first that fits is used.

Table 65 — The order that decides the default

Order	Source
1	the binding's use <source> · keep <source>
2	the name of this op's own use clause (if the type fits)
3	the **only** one among this op's inputs · bindings whose type fits
none	E-ALLOC-NOSOURCE — no allocator is visible
two or more	E-ALLOC-AMBIGUOUS — it does not guess. Write it in the binding

5 The default **does not cross op boundaries.** The source comes only from that op's signature and body, and the caller's allocator does not flow in by itself. Fields are not counted as sources either. That is, **there is no global allocator.**

5a The choice of allocator is resolved **on the tree** — after the processor has stood forms up by arity, before monomorphisation. If checked in a comparison mode that does not build the tree, unresolved clauses remain, and the processor does not silently check them as an op short of one input but says so with E-USING-UNRESOLVED.

6 The root operand can be left out, as in `alloc_bytes capacity <n>`, only **inside a region block**, and then the innermost region goes into that place. Capabilities (`cap allocator` · `cap heap`) cannot be left out and are written by name — so as not to make a way of reaching a root without a name. An `alloc_bytes` that points at no root — left out outside a region, or naming something that is none of a capability input · a region parameter · an open region block · a capability field — does not conform (`E-ALLOC-NOROOT`). The root of `stack_new` is a region parameter and cannot be left out (the same code).

6a The size is written after the `capacity` marker: `alloc_bytes <root> capacity <n>` · `stack_new <region> capacity <n>`. Without the marker it does not conform (`E-ALLOC-CAPACITY-MARK`) — that marker is what separates the shortened form from the form with a name.

In plain words

An allocator is a value — the cursor and the backing bytes have to live somewhere during execution. The `use` of an op head moves the place where that value is written **without hiding it**: the receiving side writes «I carve from this allocator» in its signature, and the caller writes «with this» on the line that makes the object. The machine code between the two is the same as passing one argument.

Example — alternating between two allocators in one op

Source

```
module ex_using .

trait carver do
  reserve input s self . input n u64 . output u64 . effects state .
end .

rem Two policies — one counts what is asked, the other twice as much
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 The receiving side — the caller writes neither the allocator nor its type
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 .
```

Rejected example — the allocator clause of an op head is `use` — tests (1b)

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 .
```

Diagnostic: `E-USING-OLD`

Rejected example — two allocators are visible and none was chosen

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 or f? — it does not guess
  return narrow u8 x. . .
end .
```

Diagnostic: `E-ALLOC-AMBIGUOUS`

9 Library

1 This clause does not specify a list of individual library functions. What it specifies is **the rules a library shall keep**. As long as those rules are kept, any library is within the guarantees of this language.

> **In plain words**
>
> The C standard holds the language and the library in one document and specifies function by function. Lowent does not, because libraries change much faster than languages. If a list of functions were written here, this document would soon be out of date. Writing **rules** instead does not go out of date.

9.1 Libraries are written in the same language

1 **The portable layer** of the standard library is written in the language this document specifies and has no privileges. That part receives the same rules as code the author writes directly.

1a But the library as a whole is not one layer. At the places that reach the outside world there are narrow leaves, and they cross a **trust boundary** (§4.4) using `unsafe` or `extern`. That boundary is kept narrow, and where it is is written in the source.

1b Turning names into addresses (DNS) is **a different leaf from making connections**. Both receive `cap net` and produce the `io` effect, but there is a reason to separate them: a program that already has an address **never touches** name resolution, and tests can measure connections **without a network** by giving addresses by hand. And name resolution can give different answers depending on the machine and the moment, and which line that wavering is on is visible in the source.

> **In plain words**
>
> For convenience, one connection leaf that also takes names would be better. But then *"how many times does this line touch the network"* could not be known from the source alone — the same question this language repeats in capabilities and effects, and it gets the same answer: **name what is hidden.**

2 That is, library ops also write contracts (§6.4), declare effects (§7.1), and receive capabilities as arguments (§7.2).

> **Note (informative)**
>
> That libraries have no privileges is a matter of checkability. If there were a hidden passage only libraries could use, the program's guarantees would break the moment that passage was crossed.

9.2 What libraries shall keep

1 a) **They return failure as a value.** They do not express failure with special values (−1 and the like) (§6.2.8).

2 b) **If destroying can fail, they do not hide it** (§8.5).

3 c) **They do not make up capabilities themselves.** Capabilities they need are received as arguments.

4 d) **They declare effects honestly**, including the effects of what they call inside.

5 e) **For numbers whose base can differ, they write which base.** For values such as rows · columns · indices · page numbers, where counting from 0 or from 1 can differ, the op's `lowdoc` or the module contract **shall state it**. This language's indices are 0-based, so libraries are **0-based by default**, and if an outside convention is 1-based (ANSI cursor positioning of terminals and the like), that conversion is done **inside the op** — if the caller had to know the outside convention, that would be knowledge outside the source (§1.3).

Note (informative)

Example: the row and column of `term.goto <buffer> <position> <row> <column>` are **0-based** (the top left of the screen is `0 0`). ANSI is 1-based, so the op adds 1 inside before putting it out — that addition is not the caller's business.

Example — libraries keep the same rules

Source

```
module ex_lib .

enum parse_error do
  too_short .
end .

rem Failure is returned as a value — no special number (-1 and the like) is used.
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 .
```

In plain words

These four are not new rules; they say that **libraries keep the rules already fixed** too. If libraries were made exceptions, the program's guarantees would break the moment such a library was used.

9.3 What is language and what is library

1 The core ops and builtin ops of §6.3.3 are **language**. Their names are fixed. The names of core ops cannot be hidden by the author, and builtin ops stand only after `call_builtin`, so they do not meet the author's names.

1a Builtin ops are material for building libraries. The standard library provides ops that wrap builtin ops, and application programs call those ops — calling builtin ops directly is the business of places that build system libraries and processors.

2 Everything else is library, used by importing modules.

3 For a feature to enter the language, the following shall hold — it cannot be expressed as a library, or expressing it as one hides a cost.

4 Between the language and libraries there is one more layer — the *leaf*.

leaf (리프)

The thinnest piece that touches the operating system or hardware. It cannot be written in Lowent, so the processor provides it. A leaf **shall be thin** — what can be written in Lowent is not a leaf but a library.

5 Whether a piece qualifies as a leaf is decided by the following question — *"can this be written in Lowent."* If it can, it is a library. **Convenience does not qualify something as a leaf.**

Table 66 — The three layers

Layer	Who makes it	Qualification
language	this document	cannot be expressed as a library, or expressing it as one hides a cost

leaf	the processor	touches the operating system · hardware. Cannot be written in Lowent
library	anyone	everything else. Written in Lowent

> **Diagram — the three layers — reading one file**

```
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
```

> **In plain words**

For example, «opening a file and receiving an integer handle» touches the kernel, so it is a leaf. Wrapping that handle in **a value that cannot be forgotten** can be written in Lowent, so it is library. «A convenience function that reads a whole file» is library too — being convenient does not make it a leaf.

> **Caution — Convenience is not a reason to enter the language**

"It is convenient to write it this way" is the business of libraries. Adding a word to the language makes that word something every program has to memorise (§6.1.2). So the language accepts only **what cannot be expressed**.

9.4 How ripe is it — the ladder of maturity

1 Each library module writes its **maturity**. There are four rungs, and a higher rung **includes all** the duties of the lower rungs.

Table 67 — The rungs of maturity and their duties

Rung	The duties of that rung
`experimental`	the maturity **is written**. Nothing more — that is why it is an experiment
`incubating`	+ a decision document that owns it **actually exists**
`standard`	+ there is a description for people to read, and a regression witness calls that module **by name**
`deprecated`	+ what to switch to is written

2 A module that does not keep the duties of the rung it writes does not conform. **Raising** a rung comes after meeting the duties; if they are not met, it shall stay on the lower rung.

> **Note (informative)**

The value of the ladder is not in the upper rungs but in the lower ones. Writing `experimental` is writing *"do not rely on this yet"* **in the source**, and that one line answers for the user *"may I design relying on this"*. Without rungs every module looks the same, and then **the least ripe is read as if it were the ripest**.

> **Note (informative)**
>
> A rung is **the size of a promise**, not a ranking of quality. A well-made experimental module can be better than a sloppy standard module — what differs is *"how much it has said it will not change"*.

9.5 What enters the standard library

1 Things enter the library **by evidence**, not **by existing**. The following six are the measures that decide entry.

Table 68 — The library charter

Measure	What it means
if it can be written in this language, write it in this language	something becomes a leaf inside the processor only when it **uses an outside authority** or is work this language cannot express. **Convenience is not a ground**
the signature is the ledger of costs and authority	effects · capabilities · the caller's storage · allocation · ownership · completion · machine constraints **shall be visible at the boundary**
separate computation from authority	building a shape and putting it out, parsing and opening a file, the next step of random numbers and obtaining entropy, protocol encoding and carrying are **each different things**
divide places along several axes	authority · storage · execution tier (⟦§10.8⟧) · profile (⟦§10.7⟧) **are not squeezed into one ladder**
keep what is inside small	up to a portable base and thin, audited host adapters. Large outside dependencies · policy-heavy facilities · surfaces still in research are not brought in
each module has evidence	it declares its support boundary and has unit · negative · oracle · failure · profile gates

2 This document **does not write a list** of libraries. What exists is decided by a ledger that is counted again from the source each time, and if this document copied it, **they would part from the moment of copying.**

> **In plain words**
>
> If the charter doubled as a list, something still a proposal would look **as if it already existed** just because it was in the table. That really happened — three APIs that did not exist were read as fact because they were in the table. So this clause specifies only **what enters**, and leaves **what has entered** to the counting side.

10 Concurrency

1 This clause specifies the places where several things **proceed together**. Concurrency is not a special feature in this language but one more place where the effect rules of §7 apply unchanged.

10.1 Concurrency is an effect too

1 Making another flow of execution, or waiting for its progress, is an effect. `wait` and `concurrent` are those (§7.1), and they shall be written in the contract.

2 What separates `wait` from `concurrent` is **who wakes it up**.

Table 69 — The two kinds of waiting

Effect	Meaning
`wait`	it can be suspended. What wakes it **is not another task of the same program** (the kernel · a device wakes it)
`concurrent`	it ends only **when a peer runs**. There has to be an executor to run that peer

> **In plain words**
>
> For example, waiting for data to come from a file is `wait` — what wakes it is not another task of the same program but the outside.

> **Note (informative)**
>
> Why this distinction matters: a small device without an operating system may have no «executor that runs peers». Then a program that uses only `wait` runs as it is, but a program that uses `concurrent` does not. It is written in the effects, so it can be known **at translation time**.

10.2 Actors

1 An *actor* is a unit of execution with its own state. The state exists only inside the actor and cannot be touched directly from outside.

1a So **reading the state fields** of an actor value **from outside** is refused (`E-ACTOR-FIELD`). The only way to reach the state is messages — put an op that returns what is needed in the actor.

1b A state field cannot be **a borrow** (`ref` · `mut ref`) (`E-ACTOR-STATE-REF`). A borrow cannot outlive what it was borrowed from (§8.4.1), but a state field remains for as long as the actor lives, and there is no place to write what was borrowed.

1c The state fields of an actor just made with `spawn actor` are **all 0**. 0 is not a slice, so **reading before setting up** a field that holds a slice or a borrow does not conform (`E-ACTOR-UNINIT`). The processor refuses the case where **the first message sent** to that actor reads such a field without setting it up — it looks within the straight order of statements.

> **In plain words**
>
> Until 2026-09-16, before (1c), that place was **where the two back ends parted**: the VM stopped with «not a slice», and native silently gave `none` (defect note 53). One of the two was a lie, but the canon did not say which. Instead of choosing an answer, it makes that place **unreachable**.

1d **A name holding an actor can start with the literal `0`** — an actor not yet spawned (the same as handle 0 of §10.3 (2d)). Other numbers do not go into an actor place (`E-TYPE-LET` · `E-TYPE-VAR`). Sending a message with `send` · `spawn send` · `try spawn send` to an actor that is 0, or `draining` it, stops the program (`E-VM-HANDLE-ZERO`). Use it after it holds `spawn actor <name>`.

Rejected example — filling an actor place with a number other than 0

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 .
```

Diagnostic: `E-TYPE-VAR`

2 Ops inside actors also follow the kinds of §6.4.1 — if it changes state it is `proc` and shall write `effects state`; if it only reads, it is `fn`.

3 Declaring it pure while changing state is refused at translation.

4 If an actor's op `panics` (§6.5.9), that actor **can be set up again.** Then that actor's state returns to the start and that message is processed again. What comes back to life is **only its own state**; other actors are not touched.

4a Setting an actor up again **does not depend on how the message came** — sent with `send`, waiting, or put in the mailbox with `spawn send` and delivered by `drain · schedule` (§10.5), it is the same. The messages **still in the mailbox** for that actor **are not touched** — the actor set up again processes them in the order they were put in.

In plain words

The remaining messages are not thrown away because `spawn send` has already returned saying it **took** them. Throwing them away silently afterwards would hide a failure. It also agrees with processing even the message that blew up again (4) — there is no reason to redo the message that went wrong and throw away the innocent ones behind it. A message that keeps blowing up is ended by the policy of (5). If a message sent against the old state has no meaning in the state returned to the start, that actor writes `failure restart never`.

5 Until when it is set up again is decided by the clause `failure restart <policy> .`. The policies are the following three and no more.

Table 70 — Actor failure policies — a closed set of three

Policy	Meaning
`max <number> [within <number> <unit>]`	sets it up again that many times. When used up, passes it upward
`never`	does not set it up again — passes it upward the first time it blows up
`always`	keeps setting it up again without counting

5b The units of `within` are a closed three, and both singular and plural are accepted — `second`/`seconds` · `minute`/`minutes` · `hour`/`hours`. Any other word does not conform.
5c If the processor cannot count the period of `within`, it **says so** (`W-NOT-YET`). It does not silently read a clause that writes a period as `max <number>`.

> **Note (informative)**
>
> Both singular and plural are accepted for reading's sake — `within 1 second` and `within 5 seconds` are written as people write. This is not **a second way of saying the same thing** (§7.1.1): the meaning is one and only the spelling differs with the number, and the processor reads both as the same number.

5a An actor has **one** failure policy (`E-ACTOR-FAILURE`). So does its mailbox policy (`E-ACTOR-MAILBOX`, §10.5). With two policies **nobody could say** which stands, and the answer would become the processor's internal affair.
6 If it blows up after using them all, that failure **is passed upward.** No actor is left silently stopped.
7 Setting up again applies only to `panic`. A broken contract (§6.4) is not set up again — that is the program breaking a promise it wrote itself, so doing it again gives the same.

> **In plain words**
>
> Why «let it blow up»? There are failures for which setting up again is better — cases where the state somehow became strange. Code that rolls back such failures one by one by hand is long, and when it is long that code itself goes wrong. Returning the state to the start is **short recovery that is always right**.

> **Example — an actor**

Source

```
module ex_actor .

actor counter do
  state do
    value u64 .
  end .

  rem It changes state, so it is `proc` and `effects state`.
  proc inc output u64 . effects state .
  do
    set value. expr value. + 1 . .
    return value. .
  end .

  rem It only reads, so it is `fn` and has no effects.
  fn get output u64 .
  do
    return value. .
  end .
end .
```

> **Rejected example — declared pure while changing state**

Source

```
module ex_actor_bad .
```

```
actor counter do
  state do
    value u64 .
  end .

  fn inc output u64 .
  do
    set value. expr value. + 1 . . rem it changes state but is an `fn`
    return value. .
  end .
end .
```

Diagnostic: E-EFFECT-PURITY

10.3 Making flows of execution

1 `spawn` makes a new flow of execution. Making one is an effect and shall be written in the contract.

2 Whether it **waits for** the flow it made **to end** or detaches it is separated by effects — detaching is `detach`.

2a `spawn <op> [arguments…]` gives a **handle** that points at that flow. The type of the handle is `task T`, where `T` is that op's output type. If it is used only as a statement without receiving the handle, that flow is detached.

2b `await <handle>` waits until that flow ends and gives its result. The type of the result is the handle's `T`. Waiting for a value that is not a `task T` is refused (`E-TYPE-KIND`). While waiting, other flows proceed — waiting does not stop the machine.

2c Waiting for something that cannot end stops the program. The processor shall say where.

2d **Handle 0 is a handle that points at no flow.** A name declared `task T` can start with the literal `0` — the spelling for leaving the place empty before receiving a flow. No other number can be a handle (`E-TYPE-LET` · `E-TYPE-VAR` — numbers do not point at flows). Waiting for handle 0 stops the program (`E-VM-HANDLE-ZERO`). The processor builds the handles `spawn` gives as values other than 0, and filters out 0 at every place that uses a handle — an empty handle **never silently points at another flow.**

Rejected example — making a handle from a number other than 0

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 .
```

Diagnostic: E-TYPE-VAR

> **Note (informative)**
>
> Handle 0 is in the same position as the null pointer of other languages. Being able to leave it empty is convenient, but using what was left empty shall **stop on the spot** rather than grab something unrelated. This language counts handles from 1 so that 0 overlaps no flow, and checks for 0 at every place it is used.

> **Note (informative)**
>
> Why waiting is an effect. `await` computes no value but **spends time** — the caller has to put that cost into its budget. So a pure op cannot wait (§7.3).

3 `spawn actor` makes an actor, and `send` sends a message to that actor. An actor processes messages **one at a time**.

3a Values can be carried with a message — written **side by side** after the message name, as in `send <actor> <message> <value>…`. On the receiving side the actor instance is treated **like the first parameter**, and the values written follow it.

3b Sending a value with ownership **hands over** that ownership. The sender can no longer use that value, and using it is refused at translation (§8.5).

> **In plain words**
>
> Why hand it over? If two flows hold the same value together, races arise. If ownership moves only through messages, at any moment only one flow can touch that value, and then races **cannot happen** — not prevented, but with no place to exist.

> **Example — making an actor and sending a message**
>
> **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 Make an actor (`spawn`) and send a message (`send`).
> export proc use_counter output u64 . effects state .
> do
>   var c counter spawn actor counter . .
>   return send c. inc . .
> end .
> ```

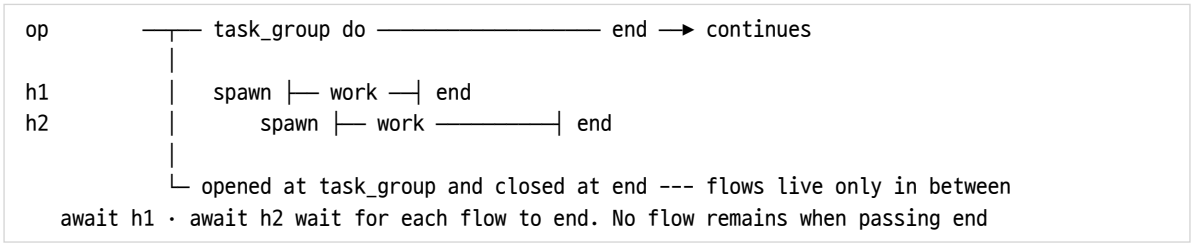
> **In plain words**
>
> It matters that an actor processes messages one at a time — the state inside **is not touched at the same time**. That is why locks need not be taken by hand.

> **Caution — Concurrency is not a world outside the checks**
>
> In many languages concurrency is the area of *"be careful from here on"*. In Lowent **the same effect rules apply unchanged** — what it waits for, whether it changes state, whether an executor is needed, are all written in the contract and checked by the processor.

10.4 The place that binds flows — `task_group`

1 `task_group do <statements> end .` **binds into one** the flows made inside it. At the place where the block ends, the bound flows have all ended, and it does not leave with even one remaining.

2 So a flow **cannot outlive the block that made it.** This is how this language handles the lifetime of flows, and detaching (§10.3 (2)) is the only exception to that discipline.

> **Diagram — the lifetime of flows is confined inside the task_group block**
>
> 
>

3 Inside a group, `spawn <op> [arguments…]` **puts a job in the queue.** It is the same word, but `spawn actor <name>` makes one actor, and the two are told apart by what follows.

4 `task_group cancel_on_error do … end .` **propagates cancellation.** If one bound flow ends with an error, the **siblings** not yet started or waiting are treated as having ended with that error.

5 Cancellation **does not roll back.** The effects of flows that already did work do not go back, and cancellation only keeps work still to come from being done. So *"which siblings were cancelled"* depends on order, and this document does not fix that order.

6 The error itself shows through `await` (§10.3 (2b)) — cancellation does not swallow errors.

> **In plain words**
>
> Why bind them? A flow made and forgotten remains after the program ends, or touches what is already gone, or produces a failure nobody reads. When the block holds that lifetime, those three disappear **by grammar** — there is no place to forget.

10.5 Delivering messages later

1 `send` (§10.3 (3)) waits until the actor processes the message. `spawn send <actor> <message> [arguments…]` does not wait but **puts it in the mailbox and returns at once.**

1a `try spawn send …` does not stop when the mailbox is full but gives that as a value — its type is `result u64 mailbox_full`.

1b If the actor that receives a waiting `send` is **handling a message held by another flow, and that flow is stopped** (by `yield` · a channel · `await` · a waiting `send`) — whether a message taken from the mailbox or another flow's waiting `send` — this `send` **waits until that message ends** and is then handled. One actor's messages go one at a time (4a). If what holds the actor is **the very chain that called this `send`** (a handler sending to itself, or coming back through A ⟶ B ⟶ A), it does not wait and is handled at once — `mailbox bounded N` counts that depth (§10.2). If flows wait on each other so that none can end, the program stops (`E-VM-DEADLOCK`).

1c `try send <actor> <message> [arguments…]` **does not wait** at a place where (1b) would wait, but gives that as a value — its type is `result <the handler's output> actor_busy`. When there is nothing to wait for, it is the same as `send`, and the handler's answer is in the `ok`.

> **In plain words**
>
> Before, the `send` of (1b) called the handler on the spot. Then the stopped message woke up and overwrote that work with the old state it had read — the sender got its answer, and the work vanished. Waiting keeps the work; a place that cannot wait receives the fact as a value with `try send`.

2 What is in a mailbox **is not processed by itself.** `drain <actor>` empties that actor's mailbox **in the order put in**.

2a If, when `drain` is called, that actor is **still handling** a message taken from its mailbox — `schedule` has taken it and it has not finished, or another `drain` is running it — the program stops (`E-VM-DRAIN-BUSY`). It neither delivers anyway nor waits.

> **In plain words**
>
> Delivering anyway would have one actor handle two messages together — exactly what (4a) prevents. Waiting would never end when it is called from inside that message's handler, and `drain` would become something that waits for a peer (`concurrent`). Stopping shows the place and keeps the meaning of `drain` as it is. Move the draining to after that message has finished — for example to the code that called `schedule`.

3 The processor does not decide by itself when to deliver because of **determinism** — the same program shall give the same answer, and the moment of delivery is the answer.

> **Caution —**
>
> If messages are only put in a mailbox and it is never emptied, those messages **are not processed.** This is not a forgotten failure but what was written — a person chooses the place that empties it.

4 `schedule .` delivers the pending messages of **all actors** until none remain. If `drain` empties one, this empties all.

4a `schedule` too handles **the messages sent to one actor in the order put in, one at a time** — the same as `drain`. Even if the handler gives up its place with `yield` (§10.6 (4)) or stops on a channel, that actor's next message starts **after** the earlier one has finished.

4b The order between messages of different actors, and between messages and tasks, is not specified by this document. Whether an answer leans on that order is checked by `schedule explore_interleavings` (§6.4.13 (4)), which runs every open order.

> **In plain words**
>
> The order inside one actor is fixed because an actor **holds its own state alone** (§10.2 (1)). If two messages to the same actor went on together, the two would take turns changing the same state, and a race would arise inside the actor — the reason for using an actor would be gone. The order between actors is left open because fixing it gains nothing and loses something: once fixed, a program that leans on that order does not show up in tests, and when the work is divided over several cores every actor would have to wait for the others.

> **Example — inside one actor, in the order put in**
>
> **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 .
```

Output

```
run() = 12345
```

10.6 Exchanging between flows — channels

> **channel** (채널)
>
> A container that passes values between flows. Values come out in the order put in, and the number it can hold is fixed. When full or empty, the flow using it stops and continues when the other side comes.

1 A *channel* is **an ordered container** into which values are put and from which they are taken between flows. It is made with `channel [<type>]`, put into with `chsend <channel> <value>`, and taken from with `chrecv <channel>`.

1a The type of `channel T` is `chan T`. The value put in with `chsend` shall fit `T` (`E-TYPE-WIDTH` · `E-TYPE-SIGN`), and `chsend` gives `u64` 0. `chrecv` gives `T`. Putting into or taking from a value that is not `chan T` is refused (`E-TYPE-KIND`).

1b For channel handles too, 0 is the empty handle (§10.3 (2d)). A name declared `chan T` can start with the literal `0`, and other numbers are not accepted. Putting into or taking from handle 0 stops the program (`E-VM-HANDLE-ZERO`).

> **Example — the types of handles and channels**

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 The handle is `task u64` and the channel `chan u64` — the types of the value a flow will give
and of the values a channel holds are carried in the type.
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 .
```

Output

```
both() = 61
```

Rejected example — receiving a handle as a number

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 .
```

Diagnostic: `E-TYPE-LET`

2 A channel **has a fixed size.** Trying to put into a full channel stops the putting flow, and trying to take from an empty channel stops the taking flow. A stopped flow **continues** when the other side comes.

3 So channel operations produce the `concurrent` effect (§7.1) — because their completion **depends on the progress of another flow**. This is a different requirement from `wait`, which the kernel wakes up someday, and that difference is carried in the signature.

4 `yield` makes the currently running flow **give way.** Stopping on a channel is built on top of this giving way.

5 If every flow stops on channels and nobody can proceed, that is **a halt**, and the processor shall say so. It does not hang silently.

> **In plain words**
>
> Why do the effects differ? An op that `waits` ends someday even if left alone, but an op that is `concurrent` **never ends without a peer.** So if there is only one bound flow and it requires a peer, that program cannot end in any order, and the processor refuses it **at translation** — it does not wait for it to run and hang.

10.7 How far it can be used — `build profile`

1 A program can write with `build profile <name> .` in which place it runs. Once written, the processor refuses **at translation** the concurrency that place cannot bear.

2 The profiles are the following four and no more; other names are refused (`E-PROFILE-UNKNOWN`).

Table 71 — Profiles and the levels they open

Profile	Level opened	What it means
`freestanding`	0	no operating system. Uses no concurrency
`embedded`	1	up to running fixed jobs divided — flows are fixed **statically**
`native`	2	makes flows and exchanges through channels
`server`	3	up to actors — flows with state live by messages

3 If what a program uses is higher than the level the profile opens, it is refused (`E-PROFILE-LEVEL`). Merely declaring an `actor` is level 3, and `spawn` and `task_group` are level 1.

4 **If no profile is written, there is no gating.** Only those who write it take on that promise — a program that does not write it has not said where it runs, and the processor does not make up what was not said.

> **In plain words**
>
> Why not judge the level by a single word? `send` is used for both actors and channels, and `drain` may be the name of an op the user made. Raising the level by looking at words alone **would accuse unrelated programs.** So level 3 is judged only by an `actor` **declaration** — if one word has two meanings, judgment is not made by that word.

10.8 What a machine can bear — `build tier`

1 If the profile of §10.7 says *"in which place it runs"*, `build tier <name> .` says *"what this machine **can bear**"*. The two are different axes — one separates the constructs used, and the other separates **effects**.

2 There are four tiers, a closed set. Other names are refused (`E-BUILD-TIER`).

Table 72 — Tiers and the effects they bear

Tier	What machine	Effects newly borne
`t0`	cooperative floor — very small machines, no operating system	`none unsafe panic state wait cancel`
`t1`	small machines	in addition to the above, `io device`

`t2`	real-time operating systems	in addition to the above, `alloc` `lock` `atomic` `blocking` `concurrent`
`t3`	machines with an operating system	in addition to the above, `heap` `page_fault` `detach` — that is, **everything**

3 Using an effect the declared tier cannot bear is refused **at translation** (`E-TIER-EFFECT`). This is not a matter of being slow during execution but of **something that cannot be loaded onto that machine at all**.

4 Effects rise along the calling chain (§7.1), so the refusal occurs **at the calling place** — not at the place where the effect first arose.

5 **If it is not written, nothing is blocked.** The default for a program that does not write it is `t3`, that is, nothing is blocked. The gate is **something chosen**.

> **In plain words**
>
> Why not merge the two axes into one? Profiles separate *constructs* (actors · flows · channels), and tiers separate *effects* (the heap · locks · atomics). There are small machines with an operating system, and large machines without one. Binding the two into one word would make machines where the two diverge **impossible to describe**. Two axes, two words.

10.9 Where running side by side goes wrong

1 Making a flow (`spawn`) can be done **only inside a binding place** (`E-SPAWN-SCOPE`, §10.4). A flow nobody waits for is not made.

2 If a binding place makes **only one** flow and that flow waits for a partner, it is refused (`E-CONC-ALONE`) — waiting without a partner never ends.

3 If all bound flows only receive and **nobody sends**, it is refused (`E-CONC-DEADLOCK`). It is a deadlock known **just by looking at what is written**, not one known only by running it, so it is stopped on the spot.

3a Sending an actor a message **that actor does not handle** does not conform (`E-IR-UNDEF`). The messages handled are looked up in that actor's declaration, and those of another actor with the same name **are not silently borrowed.**

4 Channels without an end (`channel … unbounded`) are not accepted (`E-CHAN-UNBOUNDED`). Use a channel with a fixed size.

5 The shared state types (`lock` · `rwlock` · `shared_read`) have not been built yet (`E-LOCK-NOTYET`).

> **In plain words**
>
> (4) is **a decision not to build it** (RFC-0150 Q6 ©, 2026-10-15). A channel without an end has a value whose memory use cannot be written up front, which collides with the principle that costs are visible in the text (§1.3). A channel that fills stops the sending side, and that stop is visible in the text.

> **Caution —**
>
> (3) does not catch every deadlock — it catches only those known from what is written. That there are deadlocks it cannot catch does not mean this rule is weak; it makes clear **what this rule promises** (§8.6).

A Annex A — Grammar summary

1 This annex is **reference material that gathers the parts of the grammar** specified by §6 **that are often looked up**. The norm is the main text; where this annex and the main text differ, the main text is right.

1a **The keyword list (A.1) and the full grammar (A.10) are exhaustive** — what is not in A.1 is not a keyword, and a sequence of words that the productions of A.10 cannot read is not source of this language. A.2 to A.5 and A.7 gather commonly used shapes for easy viewing and do not contain every shape.

1b What A.10 specifies is **form** only (§1.5 (4)). Which of the well-formed ones conform is decided by the clauses of the main text.

A.1 Keywords

1 The following are all the keywords. What is not in this list is not a keyword.

> **Grammar shape — the keyword list — this list is all of it**

```
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 Reserved keywords cannot be names (§6.1.2). Every keyword in this list is dealt with by this document, and there is **no** keyword that is only reserved without being dealt with.

> **Note (informative)**
>
> That this list fits on one screen is not a coincidence but a discipline (§6.1.2). To add a keyword, it has to show it is worth that much.

A.2 The shapes of statements

1 Below are commonly used shapes. <…> is a place to fill in. Declarations that have blocks are closed by the dot after `end` too (§6.1.6 (2b)).

> **Grammar shape — declarations**

```
module <name> .

use <module name> .

type <name> <type> .
newtype <name> <type> .

struct <name> do
  <field name> <type> .
end .

enum <name> do
  <variant name> .
end .

export fn <name> input <name> <type> . output <type> .
  requires <condition> .
```

```
  ensures <condition> .
do
  <statements>
end .

proc <name> input <name> <type> . output <type> . effects <effects> .
do
  <statements>
end .
```

A.3 Statements

Grammar shape — statements

```
let <name> <type> <expr> .
var <name> <type> <expr> .
let <name> <type> <expr> else <leaving statement> .
set <place> <expr> .

if <condition> do <statements> end .
if <condition> do <statements> end else do <statements> end .
while <condition> do <statements> end .
for <name> <source> do <statements> end .
repeat <name> <type> <count> do <statements> end .
range <name> <type> <first> <last> do <statements> end .
cycle <name> <type> <first> while <condition> next <next value> do <statements> end .
guard <condition> else <leaving statement> .

return <expr> .
break [<number>] .
continue [<number>] .
```

1 The `<expr>` · `<condition>` · `<source>` · `<place>` of these shapes are **forms including their own dot** — `a.` for a variable, `add a. 1 .` for a call, `7` for a literal (§6.1.5). The dot at the end of the line closes that statement. Statements with blocks are closed by the dot after `end` too, and a chain continued with `else` has one dot at the end. A `<leaving statement>` has its own dot too (`guard gt a. 0 . else return 0 . .`). The productions of these statements are in A.10.3, and their meaning is specified by §6.5.1 · §6.5.3.1 · §6.5.4.

A.4 Variants · tests · actors

Grammar shape — telling variants apart · tests · actors

```
match <value> do
  case <variant> do <statements> end .
  case <variant> do <statements> end .
end .

test <name>
do
  expect <condition> .
end .

actor <name> do
  state do
```

```
    <field name> <type> .
  end .

  proc <name> output <type> . effects state .
  do
    <statements>
  end .
end .
```

A.5 Expressions

Grammar shape — expressions

```
rem prefix — a name opens, a dot closes. There is no precedence
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 <slice> .        idx <slice> <index> .
widen <type> <value> .   narrow <type> <value> .

rem variables — a dot is attached to the name. Literals and true · false · none have no dot
<name>.

rem infix — only inside an expr island. The island is closed by its own dot too
expr <a> + <b> * <c> .
```

A.6 The lexical grammar of literals

1 This section writes again **as productions** the literals §6.1.4 specified in prose. If the prose and these rules differ, the prose wins — this section is a summary, not a new norm.

2 Notation: `{ }` is zero or more, `[ ]` optional, ¦ alternatives, and `'…'` literal characters. Upper-case names are tokens the lexer produces, and lower-case names are the pieces that make up those tokens.

Grammar shape — number literals

```
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 **A sign is a sign only when attached.** `-5` is one literal, and `- 5` is not. **Outside** an `expr` island (§6.3), a sign attached to a number is part of the value; **inside** an island, `-` is an operator. So `g 10 -3` in a form position is **two** arguments, not a subtraction. What decides that split is not spacing but **the island**.

4 **Separators come only between digits.** That is what `[ sep ] , digit` in the rules above means — they cannot come at the very front (`_1`) · the very end (`1_`) · directly after a radix mark (`0x_1`). Separators can be used in every radix and in **every part** of a floating-point number (integer part · fraction part · exponent part).

5 **`float` comes before `dec`.** Without a point or an exponent it is read as `dec`; with a point or an exponent it is `float`. There shall be digits **both before and after** the point (§6.1.4 (2)), so `1.` is not a floating-point number but the integer `1` followed by a dot that closes a form.

6 **A leading 0 is not octal.** `dec` may begin with `0`, and its value is decimal.

Grammar shape — character · string literals

```
CHAR    = [ prefix ] , "'" , ( charchar ¦ escape ) , "'" ;
STRING  = [ prefix ] , '"' , { strchar ¦ escape } , '"'
        ¦ 'r' , '"' , { rawchar } , '"' ;   (* raw string — no escapes, §6.1 (13a) *)

prefix  = 'u' ¦ 'U' ;
charchar= ? one character (UTF-8) that is neither "'" nor '\' ? ;
strchar = ? one byte that is neither '"' nor '\' ? ;
rawchar = ? one byte that is neither '"' nor a line feed ? ;
escape  = ? one in the table «Escapes — a closed set of fourteen» of ⟦§6.1.4⟧ ? ;
```

Grammar shape — text literals and multi-line comments

```
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    = ? a line that is not the closing line ? ;
linech  = ? a byte that is not a line feed ? ;
sp      = ' ' ¦ tab ¦ CR ;
nl      = LF ;
eof     = ? the end of the file ? ;
letter  = 'a'…'z' ¦ 'A'…'Z' ;
```

6a The closing line is a line that begins, **in its very first column**, with the same `tag` as the opening line, immediately followed by something that is not `letter` · `digit` · `'_'`. So lines that begin with an indented `tag`, or with a word longer than `tag`, are `line`s. The second `tag` of the rules above shall be **the same characters** as the first — this condition cannot be written by productions alone, so it is written here in words.

6b The value of a `TEXTLIT` is the bytes of the `line`s joined with the `nl`s between them. **The `nl` after the last `line` is not part of the value.** If there are no `line`s, the value is an empty byte string. Escapes in the body are not processed (§6.1.4 (29)).

6c If, on the closing line, something that is neither `nl` nor `eof` comes after `tag` and `{ sp }`, translation is refused (`E-TEXTLIT-TAIL` for `TEXTLIT`, `E-NOTE-TAIL` for `NOTE`). If, on the opening line,

something that is neither `proc` nor `rem` comes after `tag`, it is refused (`E-TEXTLIT-OPENER`). The opening line of a `NOTE` does not read what follows `tag` to the end of the line. If there is no `tag` after `note`, it is refused (`E-NOTE-TERM`) — at the same place a `TEXTLIT` gives `E-TEXTLIT-TERM`.

7 The prefixes are **a closed set of two**. Attaching any other character in front of a literal is refused at translation (`E-STR-PREFIX`). The prefix decides what the elements of the value are taken to be (§6.1.4 (27)).

8 A character literal **shall fit in one slot.** If it exceeds one element as decided by the prefix, it is refused (`E-CHAR-WIDTH`), and an empty one is refused too (`E-CHAR-EMPTY`).

> **Note (informative)**
>
> Not writing the set of escapes again here is deliberate. If the same list were in two places, the two would certainly part, and once parted nobody would know which is normative. The set is only in the one table of §6.1.4.

A.7 Head words and closers

1 A form is opened by one **name** and closed by one **closer** — a dot (§6.1.5 · §6.1.6). The following table gathers the shape of each head word. The closer is a dot on every line — forms with blocks end with `end` `.`. So the closer column is not filled in separately; the dot written at the end of the shape is it.

2 Notation: `<…>` is a place to fill in, `*` zero or more, `[…]` optional.

> **Grammar shape — head · shape · closer**

```
head                shape                                                  closer
──────────────────  ─────────────────────────────────────────────────────  ──────

module              module <name> .
use                 use <name> from "<path>" [as <alias>] .
type            type <name> <type> .
newtype         newtype <name> <type> .
struct          struct <name> do <field>* end .  (field = <name> <type> .)
enum            enum <name> do <variant>* end .  (variant = <name> [<field>*] .)
trait               trait <name> do <signature>* end . (signature = <name> <clause>*)
actor               actor <name> do <state·clause·op>* end .
state               state do <field>* end .        (inside an actor)
contract            contract <name> do <clause>* end .
fn / proc           [modifier]* fn <name> <clause>* do <form>* end .
test                test <name> [schedule <clause>] do <form>* end .
expect              expect <condition> .           (the assertion of a test block)
input               input [comptime] <name> <type> .
use (op head)       use <name> <type> .     (the allocator the op carves from)
output              output <type> .   ¦   output <name> <type> .     (a named result — a local
starting at 0, §6.4.1 (3e))
effects             effects <atom>* .
access              access <name> <mode> .
parallel            parallel <name> <mode> .
requires / ensures  requires <condition-form>* .
errors              errors <variant> [<condition-form>] .  (one error per clause)
tests               tests <name>* .
let / var           let <name> [use <name>. ¦ keep <name>.] <type> <form> [else …] .   (a dot
even when else ends in a block)
set                 set <place-form> <form> .
into                pop <stack> into <place> .   (a prelude statement form)
if (statement)      if <form> do <form>* end [else if <form> do <form>* end]* [else do <form>*
end] .
guard               guard <form> else <leaving-form> . .   (the leaving form's dot, and guard's
```

```
dot)
while               while <form> do <form>* end .
for                 for <name> [mut] <form> [if <form>] do <form>* end .
repeat              repeat <name> <type> <form> [if <form>] do <form>* end .
range               range <name> <type> <form> <form> [step <form>] [if <form>] do <form>* end .
cycle               cycle <name> <type> <form> while <form> next <form> [if <form>] do <form>*
end .
region              region <name> <kind> do <form>* end .
borrow              borrow <name> <form> do <form>* end .
return              return [<form>] .
break               break [<number>] .            (number = the depth of loops to leave · there
are no labels)
continue            continue [<number>] .
lit                 lit <type> do <field-init>* end .  (field-init = <name> <form> .)
lit (list)          lit array <length> <type> <value>… [_] .  ·  lit slice <type> <value>… .
lit (matrix)        lit matrix <dimensions> <size>… <type> <value>… [_] .
lit (slot filling)  lit array <length> <type> do <index> <value> . … [_ <value> .] end .
lit (SIMD)          lit vec <lanes> <type> <value>… [_] .
extern              extern fn/proc <name> do <clause>* end .  (body in C)
match               match <form> do <branch>* end [else do <form>* end] .   (branch = case
<pattern> do <form>* end .)
```

2a The `<clause>*` of an `fn` / `proc` head follows the one order of §6.4.1 (3a): `satisfies` · `lowdoc` · `vector` · `priority` · comptime inputs · capability and region inputs · `use` · data inputs · `output` · `effects` · `link` · `variadic` · `asm` · `asm` · `absorbs` · `reference` · `why` · `access` · `inplace` · `invalidates` · `parallel` · `reduce` · `requires` · `ensures` · `errors` · `tests` · `schedule` (`E-CLAUSE-ORDER`).

2b The body of a **block declaration** (`struct` · `enum` · `trait` · `actor` · `state` · `contract`) is opened with `do` and closed with `end` — the same one rule as the body of an `fn` and control blocks. Shapes that open with a dot like `struct <name> .`, or only break the line after the name, are refused (`E-STMT-NODO`). A newline is not a closer (§6.1.6), so a line break does not close a head.

3 This table regulates **heads and their shapes**. The shape column writes commonly used forms, and the exact norm is the main text of each clause — where the two differ, the main text wins.

4 There is no head outside this list. Heads that are not keywords (`region` · `borrow` · `into`) are prelude operations, and they too are names fixed by §9, not something that can be newly made.

A.8 Removed words

1 The following are words that were once in this language but were **removed**. Writing them is refused (`E-VOCAB-REMOVED`), and the processor says on the spot what to use instead.

Table 73 — Removed words — and what to use instead

Removed	Instead	Why
`loop`	`while true do`	it was a second spelling of exactly the same meaning
`give`	`return`	it was a second spelling of exactly the same meaning
`unit`	`void`	two spellings for one meaning — the specification and the tool called it differently
`calcop`	`fn`	a pure computation is a function. And «op» now refers only to prelude operations

`procop`	`proc`	what produces effects is not a mathematical function
`is`	(not written)	it was decoration — nobody read it, so `type h zzz u8 .` passed
`as`	(not written)	it had no weight anywhere — block names were read and **discarded**
`local`	(not written)	it is already the default — what is not `export` is not visible from outside
`to`	`field a. b .`	infix access was removed — one meaning had four spellings
`in`	`field a. b . · idx a. i. .`	the reverse spelling of `to` — it was a third spelling
`when`	(not written)	an `errors` clause takes only one error, so no marker is needed
`on`	`proc`	inside an actor block it is already a message handler. Besides, **it bypassed the pure/ procedural bit**
`fail`	`return error <variant>`	it was not in the specification's vocabulary at all — it lived only in the old interpreter
`;`	`.`	it was **the third spelling** of the closer (⟦§6.1.6⟧)
`make`	`lit`	one head for value literals — struct · array · vector · slice values all begin with `lit <type> …`
`index`	`idx`	a frequently used word made short — reading `idx a. 3 .` · writing `set idx a. 3 . v. .`
`be`	(not written)	the end of a type is decided by the number of arguments of the type grammar — it carried nothing between the name and the type. The refusal code is `E-LET-BE` (⟦§6.5.1⟧ (3a))
`.name` (head mark)	bare prefix · parentheses	it was a third spelling for opening a head — one space turned a closer into a head
`not is_some x. . .`	`is_none x. .`	two spellings of the same meaning (`E-NOT-IS-SOME`)
`union` · `select` · `any` … (bitsets · lanes)	`bitset_union` · `lane_select` · `lane_any` …	short words are left to the author (⟦§6.3.3⟧ (1e))

2 The reasons for removal were mostly **two spellings of the same meaning** or **a word that bought nothing**. A word has to be worth its place on the list, and if it is not, it comes off.

2a The old heads of type declarations — `def struct` · `def enum` · `def type` · `def newtype` — are refused too (`E-DEF-OLD`). Type declarations begin with their own keyword (§6.1.2 (3)). `def` is not a keyword, so in places not of that shape it is an author's name. `type` is also used as the kind of a type parameter (`input comptime t type .`).

3 `as` and `to` remain only as **position markers** — in `use … as <alias>` and `case <low> to <high>`. Outside those positions they are not words.

> **In plain words**
>
> Why this document records removed words. Someone who used to read old code or old writing, on meeting such a word, asks *"did I write something wrong"*. With it written here, that question ends in one line — **it was removed, and this is used instead.**

A.9 Type words

1 The words that can come in a place that writes a type are the following and no more. Names not here are names the author made (§6.2.9) or nothing.

> **Grammar shape — type words that can be used**

```
rem numbers
u8    i8    u16   i16   u32   i32   u64   i64   usize  idiff   f32   f64
u128  i128
bool  void

rem rows and aggregates
slice   array   matrix   segments   set   stack   range   vec   bitset   mask

rem what holds an answer
result   option

rem flows and channels (⟦§10.3⟧ · ⟦§10.6⟧)
task   chan

rem places and borrows
ref   mut_ref   mut   owned   region

rem a window over bytes (⟦§6.2.13⟧)
view

rem what connects to the outside
cap   mmio
fn    unsafe_fn

rem others
self
```

2 The following words **are accepted as names but have no meaning yet.** The processor says so (`W-NOT-YET`, §4.7) — it does not accept them silently.

> **Grammar shape — type words accepted only as names**

```
byte   char   str   string   bytes_view   dyn   atomic
rng   clock   device   file_system   net   tty   process
```

2a Four (`list` · `raw` · `addr` · `rng`) were moved into this group in the edition of 2026-09-17. They were listed in (1) above, but **no clause anywhere in this canon fixed their meaning**, and lowering cannot read those names either — ops with such signatures go down to the interpreter wholesale. Rather than make up a meaning, **saying that it does not exist yet** is right. Of those, `list` got its meaning from §6.2.25 on 2026-11-12 and moved to (3) below.

2b `addr` and `raw` are words whose meaning §6.9.4 decides but which are **not built yet** (`addr <name>` obtains a place, and that is the `unsafe` effect). Since 2026-11-12 the two in a type position are refused, not warned (`E-UNSAFE-UNBUILT`). And the `rng` of `cap rng` is a **kind** of capability, so it is not covered by

this clause — only `rng` standing alone as a type is a mere name. That is why `rng` was taken down from the capability-kind row of (1) above.

2c The seven `rng` · `clock` · `device` · `file_system` · `net` · `tty` · `process` are **kinds of capability**, and they have meaning only in the position `cap <kind>` (§7.2). Written alone as a type they fall into this group — the seven are in the same situation, so **they are spoken of together.** If one were spoken of while six stayed silent, readers would take that difference to mean something.

3 The following three are **shared-state types** and are refused for now. Their names are in the vocabulary because saying *"no such type"* would be false — one cannot say **your program is wrong** about a type that is in the specification.

> **Grammar shape — type words still refused**
>
> ```
> shared_read   lock   rwlock
> raw   addr
> c8   c16   c32   c64
> b8   b16   b32   b64   b128
> list
> ```

3a The last two (`raw` · `addr`) are words whose meaning §6.9.4 decides but which are not built yet (`E-UNSAFE-UNBUILT`) — they are in the vocabulary for the same reason as the shared-state types.

3b The ten after them (`c8` … `c64` · `b8` … `b128` · `list`) are words whose meaning §6.2.25 decides but which are not built yet (`E-TYPE-UNBUILT`) — they are in the vocabulary for the same reason. If the author declares a type of that name, the declaration stands.

4 `unsafe_ptr` is not a type but **a qualifier**. It attaches in front of a type like `mut` · `owned`, and coming alone it is refused.

5 **A place for address-space qualifiers is reserved.** For execution environments in which pointer size and meaning differ by space (§5.3 (4a)), such as machines with separate program and data memory or accelerators, slice and reference types have a qualifier position that writes «which address space it points at». Its spelling and meaning will be fixed when such an execution environment is brought in as a target. For now there is only one default space, and no word comes in that position.

> **In plain words**
>
> Why these three groups are written separately. *"Can be used"*, *"the name is known"* and *"does not exist"* have the reader do **different things** — use it, wait, or find another way. Lumping the three into one list would make people decide that by running the tool every time.

A.10 The full grammar

1 This section writes the grammar of one source file **from beginning to end** as productions. The notation is that of §1.5. If a clause of the main text has a rule of the same name, the one in this section is **the same down to the characters**.

2 The grammar has two layers (§1.5 (3a)). A.10.1 divides characters into tokens, and from A.10.2 on, sequences of tokens are read. White space and comments are discarded in A.10.1 and do not appear after it.

3 The shape of a source stands on the following **four rules**. The productions of this section spell those four out.

Table 74 — The four rules that stand the shape up

Rule	Content

head	every statement and declaration begins with one head word. The head decides the shape that follows
block	`do` pairs with `end.` `end` closes only its own `do`
names and dots	a name in a use position opens a form, and one dot closes the innermost open form. A variable is a form closed at once (`a.`). Statements · declarations · clauses · fields too are opened by their heads and closed by dots
word positions	a name standing in a position that makes a name · the position of a type · a word position fixed by the grammar and the tables of annex D does not open a form (it has no dot)

4 All four rules are decided **by the characters alone** — the tree stands without reading declarations. How many arguments an op takes is not material for standing the shape up but a check after the tree stands (§6.1.5 (3)). The «dot» rule has no exceptions.

A.10.1 The lexical grammar

```
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       ::= ? a character that is not newline ?
comment-line    ::= ? a line that is not the closing line ?
end-of-source   ::= ? the end of the source ?

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       ::= ? one byte that is neither a double quote nor a line feed ?
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 )
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      ::= ? one character that is not a single quote, a backslash or a line feed ?
string-char    ::= ? one byte that is not a double quote, a backslash or a line feed ?

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      ::= ? a line that is not the closing line ?
```

1 No `keyword` token begins with `rem` · `note` · `text` — those three are not keywords but the character sequences that open comments and text literals (§6.1.1 · §6.1.4).

2 If character sequences not in `keyword`, such as `"from"` · `"input"` · `"slice"`, appear as terminal symbols in the syntactic grammar below, they are `name` tokens that have meaning **only in that place** (context words). Outside that place they are ordinary names.

A.10.2 Translation units and declarations

```
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       ::= ? one item of an `asm` clause — a sequence of words and literals (⟦§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` is the shape of a declaration whose `modifier` has `"extern"` and not `"export"` — its body is in C, so the block holds only clauses (§6.9). Other `fn` · `proc` are `op-decl`.

2 The order of `clause`s is not fixed by the productions. The order is fixed by §6.4.1 (3a) (`E-CLAUSE-ORDER`).

2a The keys and values of `package-decl` and `build-decl` are specified by §5.7 · §5.8 · §6.4.7 · §10.7.

3 The individual constraints on `struct-attr` · `variant` · `signature` are specified by §6.2.7 · §6.2.15 · §6.2.19 · §6.11.

4 A `signature` is one op name and the clauses after it. A clause begins with a clause word and is closed by its own dot, so «a name that is not a clause word» coming after a clause's dot is the name of the next signature.

5 If the `term` of `requires` · `ensures` begins with the name of a contract · trait, it is a predicate on a type (`requires ordered t. . .` — §6.11). Its shape is the same as any form.

A.10.3 Statements

```
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 "."
```

A statement is opened by its head and **closed by one dot** — so are statements that end in a block (§6.1.6 (2b)). If the dot is missing, translation is refused (`E-DOT-MISSING`). Writing more dots after what is closed is refused too (`E-CLOSER-EXTRA`).

A `call-stmt` is a `term` as it is — the dot that closes that call is the statement's dot (in `print x. .` the inner dot closes `x` and the outer dot `print`).

The first `term` of a `set-stmt` is the place to put the value — a name (`i.`) or a form that gives a place (`idx a. i. . · field p. x .`).

Words that open a `statement` (`let` · `var` · `return` · `guard` …) cannot come inside an expression. `if` is a statement, not a value (`E-IF-VALUE`).

A `block` standing alone without a head is not a statement (`E-BLOCK-NOHEAD`). Names inside a `pattern` do not open forms.

An `op-decl` standing inside a block is an op visible only inside that op (§6.10).

A.10.4 Expressions

```
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     ::= ? a word of a pipe stage — a closed vocabulary (⟦§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    ::= ? a core op whose word positions are fixed by the tables of annex D — `cast <type>
<term> .` · `isa <term> <variant> .` · `size_of <type> .` and the like ?
```

An expression is one form. A `call` is opened by a name and closed by a dot — without arguments it is the value of that name (`a.`). A form inside a form closes first with its own dot, so this grammar stands up the tree of an expression **without reading declarations**. Whether the number of arguments matches the declaration is looked at after that (`E-IR-ARITY`).

A `paren-term` only wraps one form and does not change the meaning (§6.1.5 (3a)).

Forms with word positions are **a closed vocabulary**: the heads of statements, the words written in `word-form`, and the core ops whose shapes the tables of annex D write (`shaped-form`). The

arguments of any other `call` — an op the user declared — are all `terms`. Even when a type is passed (`max_of u64. a. b. .`).

4 The op name of `method` is **the first name closed with a dot**. The names and numbers before it are steps, and what comes after are arguments (`method o. inner area. w. 2 .`). The two words of `payload` are the names of a variant and a field (§6.2.19).

5 The `operator` inside an `island` is an infix operator (§6.3.2). Parentheses group one piece of the island.

6 The words of a `stage` and the shapes of their arguments are specified by §6.12.

A.10.5 Types

```
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     ::= ? other type shapes ⟦§6.2⟧ specifies (`mask` · `set` · `stack` · `fn` · `unsafe_fn` ·
`mmio`) ?
matrix-size    ::= number ¦ name
```

> **Note (informative)**
>
> The `name` of `matrix-size` is used only in an op's `input` — it is the name of a translation-time value input of that op (§6.2.6.1 (3a)).

1 Type words such as `u8` · `bool` · `void` (A.9) and the names of types the author made are `names`.

2 A type **ends by its characters alone** — there is no closing mark. Constructors are a closed vocabulary (`slice u8` is two words, `result u64 perr` three), and any other type is one name. User types that take arguments, and shapes whose number of arguments is not fixed, are wrapped in parentheses ((`box u64`) — §6.5.1 (3b)).

3 Names and numbers inside a type do not open forms — they have no dot (`array n u8`).

4 Constructors that take a count (`array` · `vec` · `matrix`) write the count first and the element type at the very end. So even if the element is itself a constructor, no parentheses are needed — `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).

> **Note (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.

> **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.

> **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.

> **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.

> **Note (informative)**
>
> 이 색인은 **생성된 것**이다. 사람이 손으로 고치지 아니한다.

Table 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.

> **Note (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).

Table 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).

Table 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.

Note (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.

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.

> **Note (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.

Table 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:** —

Table 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:** —

Table 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:** —

Table 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:** —

Table 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:** —

Table 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`

Table 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:** —

Table 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:** —

Table 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:** —

Table 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:** —

Table 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:** —

Table 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:** —

Table 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`

Table 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:** —

Table 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 **free choice** 과 다르다(부록 F). 모호는 **정해야 하는데 아직 정하지 않은 것**이고, 뒤엣것은 **열어 두기로 정한 것**이다.

> **Note (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.

Table 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 **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.

> **Note (informative)**
>
> 이 목록은 **생성된 것**이다(`docs/spec/FREE-CHOICE.tsv`). 번호는 다시 쓰지 아니한다.

3 모두 4 줄이다.

Table 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