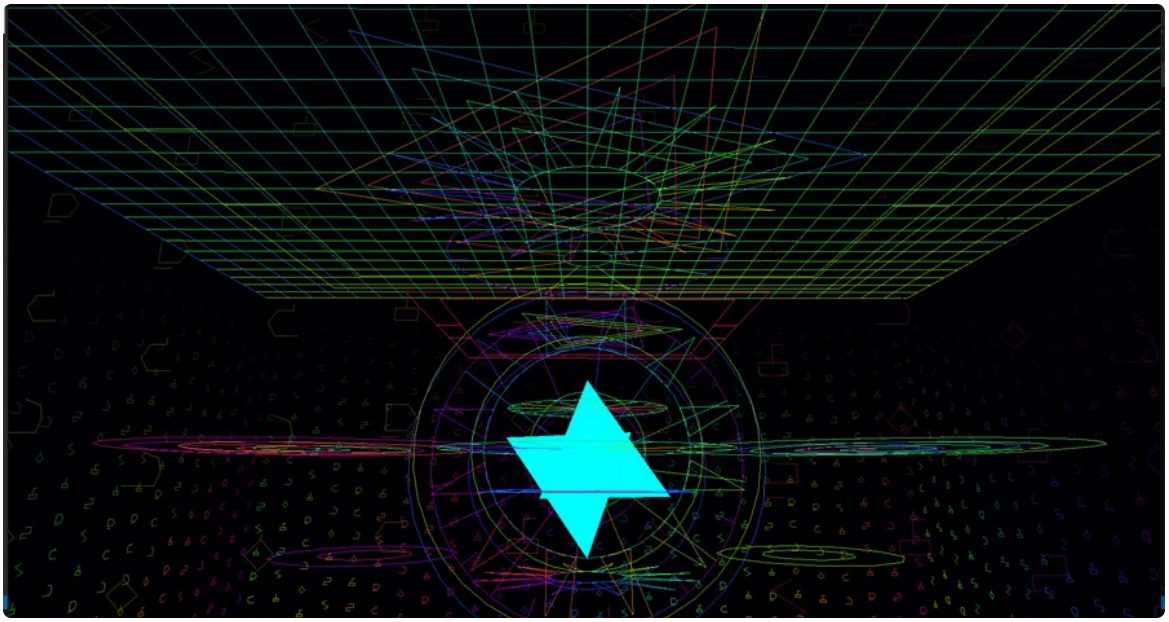
MASTERING LING



灵 · 靈 · 령 · āɴ

Mastering Ling

Advanced Edition

LING PRESS

Mastering Ling: Advanced Edition

Copyright © 2026 Ling Press. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without prior written permission of the publisher, except for brief quotations in reviews and for the example programs, which may be used freely in your own projects.

Ling and the Ling logo are trademarks of their owner. All other trademarks are the property of their respective owners. Every example in this book was written against Ling 2030.1.53. Later versions may differ; consult the official documentation at ling-lang.org.

First edition, 2026.

ISBN: TBD

Contents

WELCOME 5

Preface 6

How to Use This Book 10

PROFESSIONAL LING 15

1. How Ling Runs 16
2. Concurrency and Isolates 36
3. Network Services in Ling 59
4. Web Apps with the Rust Crates 79
5. Cryptography in Practice 98
6. Peer-to-Peer and Protocols 129
7. AI and Tensors 155
8. Using the Crates from Rust 188
9. Extending Ling 214
10. Ling OS and Embedded 238
11. Architecting Large Apps 254

RECIPES 281

12. Professional App Recipes 282
13. Music Recipes: A DAW in Ling 349
14. Crypto Recipes: A Privacy Wallet in Ling 381
15. Tooling Recipes: Editor Apps 399

SHOWCASE 431

16. Case Studies: Applications Built in Ling 432

Known Limitations and Gotchas 448

PART

Welcome

What this edition assumes and how to use it.

Preface

This book teaches you to program, using a language called **Ling**. You do not need to know anything about computers beyond how to open a folder and type in a text file. If you have programmed before, you will fly through the first chapters and find the later ones, which are about graphics, games and sound, a good deal of fun.

Who this book is for

I wrote the Beginner Edition with four kinds of reader in mind.

- **The complete beginner.** You are curious, perhaps a little nervous, and you have never written a program. Every idea in this book is introduced with an everyday comparison before it appears in code, and every program is small enough to type in by hand.
- **The teenager (or the parent of one).** You want to make games and pictures that move, not spreadsheets. By Chapter 6 you will have a 3D room on your screen, and by Chapter 7 a game you can play and show to friends.
- **The artist or musician.** You care about colour, shape, rhythm and sound, and you would like to bend a computer to your imagination. Ling draws patterns and plays music from nothing but numbers, and the later chapters explore that.
- **The hobbyist coming from elsewhere.** You may know Python, JavaScript or Scratch. You will recognise most of the ideas, and the book is honest about the places where Ling is different (there is no assignment statement, `for` loops have a surprising rule, and there is no dictionary type yet).

What is Ling?

Ling (灵言, "spirit language") is a programming language with an unusual feature: it is **polyglot**. Its keywords can be written in English, Chinese, Japanese, Korean, Thai and many other human languages, even mixed inside one file. The same program can be read by a student in Beijing, Seoul, Tokyo, Bangkok or Boston. In this book we write the keywords in English, but you will see the other spellings in the first chapter, and you can use any of them yourself.

Ling is also a practical language. A single, small command, `ling`, runs your program. The same language opens windows, draws 2D and 3D pictures, plays and analyses sound, talks to networks and builds programs that you can give to other people. It runs on Windows, macOS and Linux.

Ling is also a **young** language, and it is still growing. Some features that you may expect from older languages are missing, and a few behave in ways that will surprise you. A good book should not hide that. Where Ling has rough edges, this book tells you plainly, tells you why, and shows you how to work around them. I would rather you learn the truth than be puzzled later.

The three editions

Ling's book comes in three parts, and each is a complete book in its own right.

Edition	For	Covers
Beginner (this book)	people who are new to programming	the language basics, your first 3D room, a small game, sound and music, and how to build a project
Intermediate	people who have finished this book	data and modules, files, user-interface applications, larger games, packaging, and a collection of recipes
Advanced	experienced programmers	concurrency, networking, cryptography, artificial intelligence, compiler internals and embedding Ling in Rust

The Reference, a large catalogue of every keyword and built-in function with a verified example, accompanies all three editions.

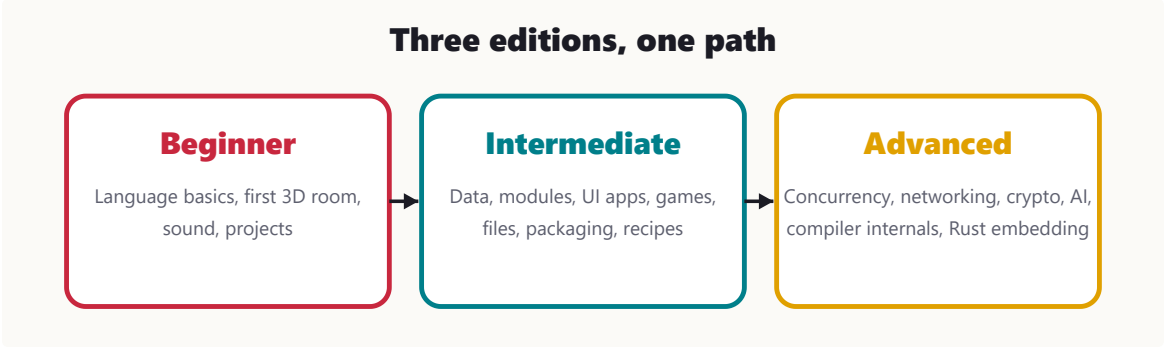


Figure: The Beginner edition covers the language basics, your first 3D room, sound and projects. Intermediate covers data, modules, UI apps, games, files, packaging and recipes. Advanced covers concurrency, networking, crypto, AI, compiler internals and Rust embedding.

How this book is organised

The nine tutorial chapters build on each other.

1. **Welcome to Ling** installs the language and runs your first program.
2. **Values and Bindings** teaches numbers, text, names and arithmetic.
3. **Decisions and Loops** teaches `if`, `while`, `for` and `match`.
4. **Functions** teaches how to package your code into reusable pieces.
5. **Lists and Text** teaches collections and string handling, with three small projects.
6. **Your First 3D Room** opens a window and builds a room you can walk around.
7. **Drawing and Games** teaches 2D drawing, keyboard and mouse, and a complete game.
8. **Sound and Music** makes sound and music out of numbers.
9. **Your First Project** shows how to organise, debug and ship a program.

Chapters 1 to 5 use only the terminal and need no special hardware. Chapters 6 to 8 open windows and play sound, so you will need a computer with a screen and speakers or headphones.

How to get the code and the examples

Every program in this book is printed in full. You can type it in, or copy it from the page, and run it. Typing the first few by hand is a good idea: it forces you to read every character.

Every example was run on **Ling 2030.1.53**, the version current when this book was written. The output printed below each program is the real output we obtained, not a hand-written guess. More example programs, from tiny scripts to full 3D scenes, are part of the Ling project's source repository at **github.com/taellinglin/ling** and the project website, **ling-lang.org**.

Conventions

A short guide to the visual conventions in the book follows in the next section, "How to Use This Book". In brief:

- `code in this typeface` is Ling code, or the name of something you type or see on the screen.
- Programs are shown in boxed listings, with their output in a separate panel just below.
- Four kinds of callout box mark **Tip**, **Note**, **Warning** and **Try it** passages.

- Every chapter ends with a summary and with exercises whose answers are at the very end of the chapter.

A last word before you begin

Programming is not magic, and it is not only for mathematicians. It is a craft, like cooking or carpentry. You get good at it by making things, making mistakes, and fixing them. The mistakes are not a sign that you are doing it wrong; they *are* the learning. Every programmer you admire has seen the same error messages that you will.

Be curious. Change things. Break things on purpose to see what happens. When something works, make it a little more ambitious. And have fun: the best reason to learn to program is that it is a delight to make a machine do what you imagined.

Welcome to Ling.

How to Use This Book

This short section explains the visual conventions used throughout the book, so that you can tell at a glance what you are looking at.

Code listings

Programs appear in boxed listings with syntax colouring. There are three kinds, each with a different label.

Complete programs

A listing that is a complete program, which you can save in a file ending in `.ling` and run with `ling run`, looks like this:

```
bind start = do {
  println("Hello, world!")
}
```

```
OUTPUT
Hello, world!
```

The panel under the listing, labelled *output*, shows exactly what the program printed when we ran it. If your output differs, check your typing first; the programs in this book are tested.

Unless a listing says otherwise, it was run twice: with the default engine, and with the careful interpreter (`ling run --interp`). When the two engines give different answers, the text beside the listing says so and explains why.

Windowed (GUI) programs

Programs that open a window, draw pictures or play sound carry a **GUI** badge on the listing:

```
GUI
bind start = do {
  open_window(320, 200, "Hello window")
  while window_is_open() {
    fill(20, 20, 60)
    set_color(255, 200, 80)
    draw_disc(160, 100, 50)
    present()
  }
}
```

A GUI listing is also a complete program, but its real result appears on your screen, not in the terminal, so usually there is a screenshot beside it instead of an output panel. Close the window or press Escape to stop it. If a GUI program also prints text, for example when it ends, an output panel is shown.

Fragments

Sometimes we show only a few lines to make a point. These are *fragments*: they are not complete programs, and they will not run unless you put them inside `bind start = do { ... }` (or inside a function). Fragments carry no output panel:

```
FRAGMENT
bind score = score + 10
```

Terminal sessions

Commands you type in a terminal and the text the computer prints back are shown in a plain panel. A dollar sign (`$`) marks the line you type; do not type the dollar sign itself:

```
OUTPUT
$ ling run greet.ling
Hello, world!
My name is Ling.
2 + 2 = 4
```

Callouts

Boxes in the margin of the text call attention to four kinds of thing.

> **TIP**
>
> A shortcut, a good habit, or a way to save time. You can skip tips, but you will probably wish you had read them.

> **NOTE**
>
> Extra background or a detail that makes an idea clearer. Notes often explain *why*.

> **WARNING**
>
> A trap. Warnings describe mistakes that cost many people time. Read these carefully; they are the product of other people's pain.

> **TRY IT**
>
> A small experiment for you to run right now. Programming is learned by doing, so please do these.

Chapter structure

Every chapter has the same shape.

1. **What you'll learn**, a short opening paragraph.
2. Short sections that each teach one idea, with small programs.
3. A **Chapter summary**, a list of the main points. It is a good way to review.
4. **Exercises** with answers under the heading *Answers*. Try each exercise before you read its answer; there is often more than one correct way.

Figures

Figures are pictures with an italic caption beneath them. Screenshots in the book were captured from real runs of the programs shown, using the `screenshot` function that you will learn in Chapter 6. Diagrams, such as the frame loop, show how an idea fits together.

Version note

Every example in this book was run on **Ling 2030.1.53** (and on `lingfu` 2030.1.19, the project tool). Ling is under active development. Later versions may add features or fix some of the rough edges that the book points out. The programs here should keep working, but the output of a newer version might differ in small ways, and a "known trap" described here may have been repaired. When in doubt, the first line printed by the bare command `ling` shows which version you have.

A few built-in functions that exist only in newer builds of Ling than the one we used (for example `str_to_hex`, `hex_to_str`, `rand_seed`, `net_local_ip`, `clipboard_get` and `clipboard_set`) are not used in the Beginner edition for that reason.

Spelling and language

The book is written in English, and Ling's keywords are shown with their English spellings. The other spellings, in Chinese, Japanese, Korean, Thai and other languages, are listed in the Reference. Where a program in this book prints a message in another script, that is intentional.

Typing the programs

A few practical points, if you are typing programs in by hand:

- Indentation is for people; Ling does not care how far a line is indented. But keep it neat. Two spaces per level makes a program much easier to read.
- Keep the quotation marks straight. A word processor often turns straight quotes into curly ones, which Ling cannot read. Use a plain text editor or a code editor.
- Copy a program exactly, including brackets. When a program does not work, compare it character by character with the book.
- Ling is case-sensitive: `Println` is not `println`.

Now turn the page and begin.

PART

Professional Ling

How Ling runs, concurrency, services, cryptography, AI, the Rust crates, extending the language and architecting large applications.

CHAPTER 1

How Ling Runs

What you'll learn

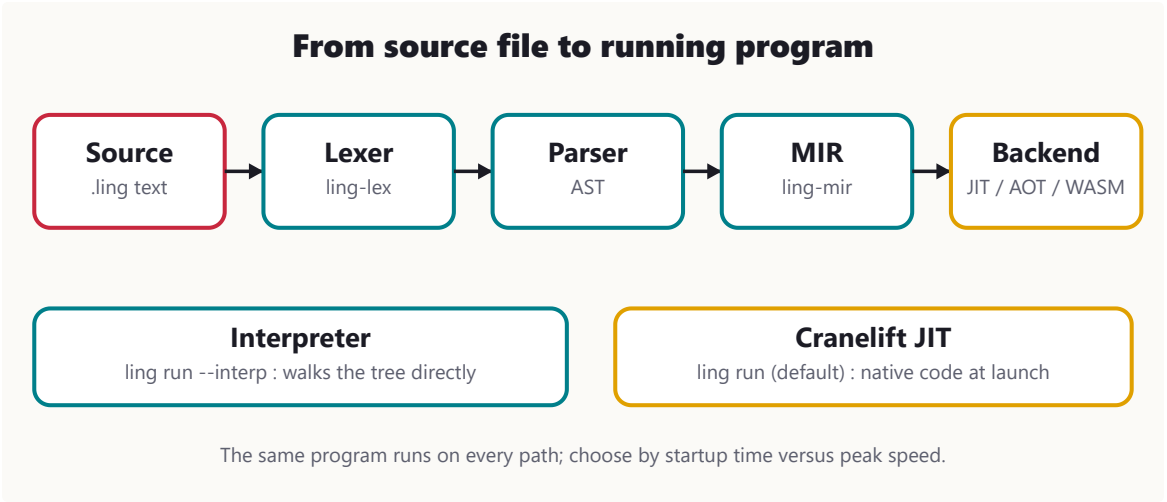
Most languages hide their execution model from you until something goes wrong. Ling is young enough that the execution model is part of your daily life: the same file can run on a native-code JIT or on a tree-walking interpreter, and in a handful of corner cases the two do not agree. This chapter gives you the map.

You will learn how a `.ling` file travels from text to a running program, what `ling run` decides on your behalf, exactly where the JIT and the interpreter diverge and how to write code that is correct on both, how to look at your program as a picture with `ling visualize` and `ling ast`, how to read the optimised intermediate representation, and how to benchmark honestly. Everything below was run on the installed `ling` (2030.1.53); where the repository source differs from the installed binary, that is said so.

The pipeline

A Ling program passes through the same front half no matter how it ends up running.

1. **Lexing.** A hand-written lexer turns text into tokens. This is where the multilingual keywords are resolved: the lexer, not any lexicon file, decides which spelling means `bind`, `choose` or `return`.
2. **Parsing.** A recursive-descent parser builds the AST. Top level may contain only `bind`, `fn`, `mod`, `form`, `choose`, `use` and `type`; a bare `println(...)` at file level is a parse error.
3. **Lowering.** Modules and `use` are flattened and the AST is lowered to MIR, a small control-flow-graph form in the `ling-mir` crate. An optimiser pass runs on it at levels O0 to O3.
4. **Execution.** Either a Cranelift JIT turns the MIR into machine code, or the interpreter walks the tree directly.



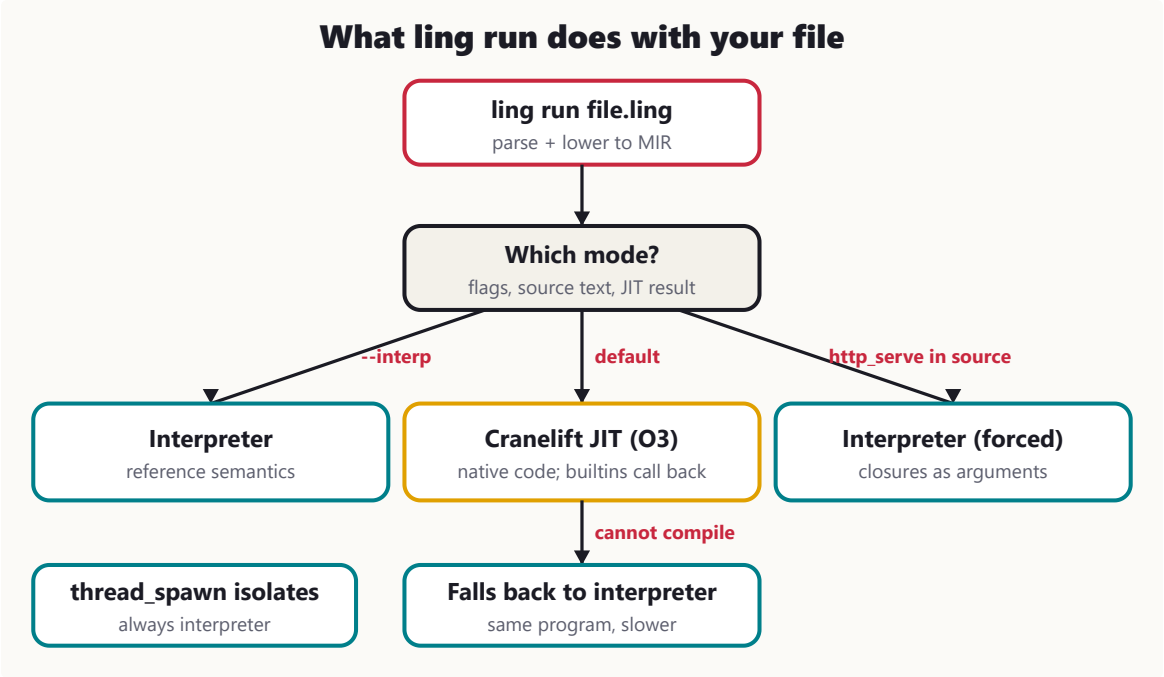
The toolchain pipeline. Every path shares the lexer and parser; they diverge at the backend.

Three facts follow from how the pieces are wired, and they explain most of the surprises later in the chapter.

- **The interpreter is the language.** Every builtin lives in one very large `match` on the function name inside the interpreter. The JIT does not reimplement builtins: compiled code calls back into the same dispatcher. A new builtin therefore works on both backends the moment it exists.
- **Compiled code is only the skeleton.** Arithmetic, comparisons, loops and calls between your own functions become native instructions. Everything else (`println`, `len`, `file_hash`, `sha256_hex`) is a call back into the runtime.
- **There is no static checker on the `ling run` path.** The repository contains a type checker and a borrow checker, but nothing in `ling run` calls them. Type annotations parse and are ignored, and the ownership keywords (`own`, `lend`, `move`, …) are accepted and not enforced. A program is never rejected for getting them wrong.

What `ling run` chooses

`ling run file.ling` picks a backend using a short, fixed set of rules. The figure shows them; the table gives the flags.



Mode selection. The default is the JIT, with an automatic fallback to the interpreter.

Command	Backend
`ling run f.ling`	Cranelift JIT at O3; falls back to the interpreter if the program cannot be compiled
`ling run --interp f.ling`	Tree-walking interpreter, the semantic reference
`ling run --O0` … `--O3`	JIT at a chosen MIR optimisation level (useful if O3 compile time explodes on one giant function)
`ling run --wasm f.ling`	Builds a web bundle and serves it; the browser runs the interpreter compiled to WebAssembly
`ling build f.ling --platform ...`	Distributable; `--aot` compiles through Cranelift object files

Two further rules are not flags.

- **Servers force the interpreter.** If the source text contains `http_serve` or `lingtp_serve` (or their localised names), `ling run` uses the interpreter regardless of flags, because request handlers are closures passed as arguments and the JIT currently lowers closure arguments to `()`.
- **Isolates always use the interpreter.** Code started by `thread_spawn` runs on its own interpreter, even if the parent is JIT-compiled. The consequences are the subject of the next chapter, *Concurrency and Isolates*.

Run the same file both ways and you can see the gap in raw speed.

```
fn fib(n) {
  if n < 2 { return n }
  return fib(n - 1) + fib(n - 2)
}

bind start = do {
  bind t0 = time_now()
  println("fib(27) = ", fib(27))
  println("seconds: ", time_now() - t0)
}
```

```
OUTPUT
fib(27) = 196418
seconds: 0.0018954277038574219
```

With `--interp` the same program prints:

```
OUTPUT
fib(27) = 196418
seconds: 0.2874925136566162
```

The timings vary run to run, but the ratio does not: this call-heavy recursion is roughly two orders of magnitude faster in compiled code. Remember that the first number comes from native code and the second from a tree walker.

> **NOTE**
>
> The WebAssembly path does not translate your code to WebAssembly instructions. It ships the interpreter, compiled to `wasm32`, together with your source. Behaviour in the browser is interpreter behaviour. The `ling build` platform builds drive a Rust toolchain (cargo) and a linker; they are not something to run casually in a scratch folder.

JIT versus interpreter: where they differ

Because the default is the JIT but the reference is the interpreter, a program can pass on one and misbehave on the other. The pattern is consistent: **the JIT fails quietly with `()` where the interpreter raises an error.** That makes JIT bugs harder to see, not rarer.

```
bind start = do {
  println("pi() = ", pi())
  println("1/0 = ", 1 / 0)
  println("empty text truthy? ", if "" { "yes" } else { "no" })
  bind xs = [1, 2, 3]
  println("xs[99] = ", xs[99])
  println("missing fn = ", no_such_function(1))
}
```

```
OUTPUT
pi() = 3.141592653589793
1/0 = ()
empty text truthy? yes
xs[99] =
missing fn = ()
```

On `--interp` the same file prints three lines and then stops with an error:

```
OUTPUT
pi() = 3.141592653589793
1/0 = inf
empty text truthy? no
error[runtime]: index 99 out of bounds
 --> div1.ling
  traceback (deepest call last):
    0: start
```

Read the two outputs side by side. Division by zero is `()` on the JIT and `inf` on the interpreter. An empty string is truthy on the JIT and falsy on the interpreter. An out-of-range index is an empty value on one and a halting error on the other. A call to a function that does not exist is `()` on one and an error on the other. A typo in a function name can ship silently in a JIT-only test.

The next divergence is about mutation.

```
bind start = do {
  bind a = [1, 2, 3]
  bind b = list_push(a, 4)
  println("a = ", a)
  println("b = ", b)
}
```

```
OUTPUT
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
```

On the interpreter, `a` stays `[1, 2, 3]`. `list_push` is documented as returning a new list, but the JIT may append in place and hand back the same storage. Never depend on either behaviour: always rebind (`bind a = list_push(a, 4)`) and never keep an alias to a list you are about to extend.

Closures are the third area, and the least predictable.

```
fn inc(x) { x + 1 }
bind start = do {
  println((|x| x + 1)(5))
  bind r = try(| | 1 + 1)
  println(r)
  println(list_map([1, 2, 3], inc))
}
```

```
OUTPUT
6
Err(try: first argument must be a function)
()
```

Interpreter output for the same file:

```
OUTPUT
6
Ok(2)
[2, 3, 4]
```

A closure called on the spot works everywhere. A closure or function value passed as an argument to a builtin (`try`, `list_map`, `list_filter`, `list_find`) works on the interpreter and degrades to `()` on the JIT. Calling a closure held in a variable errors on the interpreter and is unreliable on the JIT.

Scoping has a quieter split. In this program a `bind` inside a `match` arm persists on the JIT and is discarded on the interpreter:

```
bind start = do {
  bind n = 0
  match n {
    0 => { bind n = 99 }
    _ => { bind n = 5 }
  }
  println("after match: ", n)
  if n == 0 { bind n = 7 }
  println("after if: ", n)
  bind k = 0
  while k < 3 { bind k = k + 1 }
  println("after while: ", k)
  for i in 0..3 { bind n = n + 100 }
  println("after for: ", n)
}
```

```
OUTPUT
after match: 99
after if: 99
after while: 3
after for: 99
```

On `--interp`:

```
OUTPUT
after match: 0
after if: 7
after while: 3
after for: 7
```

Note the lines that agree. `while` and `if` bodies share the enclosing scope. A `for` body gets a fresh copy per iteration, so `bind n = n + 100` never reaches the outer `n` on either backend; that is a language rule, not a divergence. Only the `match` arm differs, and a loop that updates its flag from inside a `match` arm will terminate on one backend and spin forever on the other. The next chapter shows exactly that bug biting a channel-reading loop.

> **WARNING**
>
> The JIT is the default, so a program you only ever run with plain `ling run` can depend on JIT-only behaviour without you noticing. Anything the JIT does by silently producing `()` is a bug waiting for the day you run on the interpreter, in a thread, in the browser, or in a server.

Writing code that is correct on both

You do not need to memorise a divergence table. Adopt a small discipline and run the check below.

1. **Express control flow with `while`, `if` and function calls.** Do not rebind variables inside `match` arms; use `match` as an expression that yields a value, then rebind outside it.
2. **Rebind every list you change.** `bind xs = list_push(xs, v)`, `bind xs = list_set(xs, i, v)`.
3. **Guard what the interpreter would reject.** Check indexes against `len(xs)`, divisors against zero, and files with `file_exists` before `read_file`.
4. **Compare text with `==` only.** `"a" < "b"` is `false` on the JIT and an error on the interpreter, so sort with numbers.
5. **Do not pass function values to builtins.** Write the loop yourself and call a named function directly.
6. **Use `pi()`, not `pi`; use `len(xs)`, not `list_len`.** Unknown names are `()` on the JIT.
7. **Return failure as `bad(...)`, not as a surprise.** Callers unwrap with `match`.

Here is a small program written to that discipline:

```
form Sample { label, value }

fn safe_div(a, b) {
  if b == 0 { return bad("division by zero") }
  ok(a / b)
}

fn at(xs, i, fallback) {
  if i >= 0 && i < len(xs) { xs[i] } else { fallback }
}

fn value_or(r, fallback) {
  match r { ok(v) => v, bad(e) => fallback }
}

fn mean(samples) {
  bind total = 0
  bind i = 0
  while i < len(samples) {
    bind total = total + samples[i].value
    bind i = i + 1
  }
  safe_div(total, len(samples))
}

bind start = do {
  bind rows = [Sample("a", 4), Sample("b", 10), Sample("c", 7)]
  println("mean:    ", mean(rows))
  println("empty:   ", mean([]))
  println("at(5):   ", at(rows, 5, "none"))
  println("label:   ", at(rows, 1, "none").label)
  println("fallback:", value_or(mean([]), -1))
  bind rows = list_push(rows, Sample("d", 1))
  println("count:   ", len(rows))
}
```

```
OUTPUT
mean:    Ok(7)
empty:   Err(division by zero)
at(5):   none
label:   b
fallback:-1
count:   4
```

It prints exactly the same text under `--interp`. Notice what the discipline buys you: `mean([])` returns a visible `Err(...)` rather than `()` on one backend and `inf` on the other, and the `at` helper turns an out-of-range read into an explicit fallback.

To make the check mechanical, keep a two-backend runner in your project. This shell script exits non-zero when the backends disagree and prints a diff:

```
#!/bin/sh
# usage: dualrun.sh program.ling   -- exit 1 when the two backends disagree
ling run "$1" > .jit.out 2>&1
ling run --interp "$1" > .int.out 2>&1
if cmp -s .jit.out .int.out; then
  echo "OK: $1 behaves the same on JIT and interpreter"
else
  echo "DIVERGES: $1"
  diff -u --label jit --label interp .jit.out .int.out
  exit 1
fi
```

Run on the discipline program, then on the `list_push` demo from earlier, it prints:

```
OUTPUT
OK: portable.ling behaves the same on JIT and interpreter
DIVERGES: div2.ling
--- jit
+++ interp
@@ -1,2 +1,2 @@
-a = [1, 2, 3, 4]
+a = [1, 2, 3]
 b = [1, 2, 3, 4]
```

> **TIP**
>
> Make `dualrun.sh` part of your test target. It costs one extra run per test and catches the whole class of silent-`()` bugs before they reach production. Programs that cannot run on the JIT at all (they pass closures to builtins by design) should be tested with `--interp` only; mark them clearly.

> **TRY IT**
>
> Take any program of yours that uses `list_map` or `try` and run `dualrun.sh` on it. If it diverges, rewrite the closure as a `while` loop that calls a named function and run it again.

Seeing your program: `ling visualize` and `ling ast`

Two commands draw your program. They are the quickest way to answer "what does this file actually contain?" in a codebase you did not write.

`ling visualize file.ling` prints an SVG to standard output, so redirect it:

```
ling visualize shapes.ling > shapes.svg
```

This is the program used for the figure below:

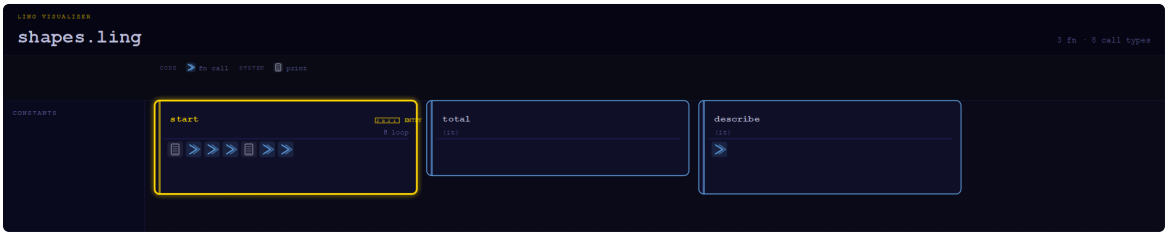
```
form Item { name, qty, price }

fn total(it) { it.qty * it.price }

fn describe(it) {
  if total(it) > 100 {
    "large order"
  } else {
    "small order"
  }
}

bind start = do {
  bind it = Item("bolt", 120, 0.25)
  bind i = 0
  while i < 3 {
    println(describe(it), " ", total(it))
    bind i = i + 1
  }
}
```

```
OUTPUT
small order 30
small order 30
small order 30
```



Real output of `ling visualize`, rasterised. Each function becomes a card; the entry point (`start`) is outlined in gold and tagged ENTRY.

Be clear about what you are looking at. This is an architectural overview, not a syntax tree: each card is a function, and the small icons inside it are its calls, with `print` shown separately from ordinary function calls. The header counts functions and distinct call types

("3 fn, 8 call types"), and a loop badge appears on `start` because it contains a `while`. It is excellent for spotting an oversized function or a function with a surprising number of calls. It is not a substitute for reading the code.

`ling ast` works on a whole project instead of one file. It scans the current directory, prints a one-line summary such as `ling ast: 'myproject' - 1 file(s), 1 fn(s)`, and writes three renderings into an `AST/` folder: a `*.technical.svg`, a `*.artwork.svg` and a `*.ling.svg`. Open them in a browser. Run it from a project folder (for example the one scaffolded by `lingfu new`), not from your home directory.

> **NOTE**
>
> SVG files embed web fonts by name and some rasterisers ignore them. If you need a PNG for documentation, render the SVG in a browser or with a tool such as ImageMagick; that is how the figure above was produced.

Reading the optimised MIR

When you want to know what the optimiser did to a hot function, ask for the MIR. The `lingc` tool (installed alongside `ling`) prints it:

```
lingc mir ast1.ling -O 3
```

For this input program:

```
fn double(n) { n * 2 }

bind start = do {
  bind x = double(21)
  if x > 40 {
    println("big ", x)
  }
}
```

```
OUTPUT
big 42
```

the unoptimised `__main__` function (abridged) shows a call to `double`, a comparison and a conditional jump:

```
OUTPUT
  bb0:
    Assign(Local(1), Call { func: Constant(Function("double")), args:
[Constant(F64(...21...))] })
    Assign(Local(4), BinaryOp(Gt, Copy(Local(1)), Constant(F64(...40...))))
    term: SwitchInt { discr: Copy(Local(4)), targets: [(1, BasicBlockId(2))],
otherwise: BasicBlockId(3) }
```

At `-O 3` the whole function collapses to a single call with the answer already computed:

```
OUTPUT
  bb0:
    Assign(Local(6), Call { func: Constant(Function("println")), args:
[Constant(Str("big ")), Constant(F64(...42...))] })
    term: Return
```

The call to `double`, the comparison and the branch were folded away at compile time. (Numeric constants print as raw bit patterns of an `f64`; the `...` above stands for those digits.) This matters when you benchmark, as the next section explains. Only `println` survives, because it is a builtin with a side effect.

> NOTE
>
> `lingc compile` (the bytecode VM) is experimental: on a plain recursive Fibonacci it failed with a codegen error while `ling run` worked. Treat `lingc mir` as the reliable `lingc` feature.

The performance model

Think of a Ling program as two layers.

- **Native layer.** Arithmetic, comparisons, `if` / `while`, and calls between your own functions. On the JIT these are machine instructions.
- **Runtime layer.** Every builtin: string handling, list operations, file and crypto calls. Both backends pay a dispatch cost here, and the JIT pays a small extra crossing cost.

A benchmark that exercises each layer separately shows it:

```
fn fib(n) {
  if n < 2 { return n }
  return fib(n - 1) + fib(n - 2)
}

fn bench_fib() {
  bind t0 = time_now()
  bind r = fib(30)
  println("fib(30)=", r, " in ", round((time_now() - t0) * 1000), " ms")
}

fn bench_builtin() {
  bind t0 = time_now()
  bind i = 0
  bind s = 0
  while i < 200000 {
    bind s = s + len(str(i)) + floor(sqrt(i))
    bind i = i + 1
  }
  println("builtin loop=", s, " in ", round((time_now() - t0) * 1000), " ms")
}

fn bench_list() {
  bind t0 = time_now()
  bind xs = []
  bind i = 0
  while i < 20000 {
    bind xs = list_push(xs, i)
    bind i = i + 1
  }
  println("list len=", len(xs), " in ", round((time_now() - t0) * 1000), " ms")
}

bind start = do {
  bench_fib()
  bench_builtin()
  bench_list()
}
```

```
OUTPUT
fib(30)=832040 in 4 ms
builtin loop=60617370 in 86 ms
list len=20000 in 3 ms
```

And with `--interp`:

```
OUTPUT
fib(30)=832040 in 1334 ms
builtin loop=60617370 in 155 ms
list len=20000 in 9 ms
```

Pure computation (`fib`) is more than 300 times faster compiled. The builtin-heavy loop is only about 1.8 times faster, because most of its time is inside the runtime's `len`, `str` and `sqrt` dispatch, which both backends share. The practical rule: **the JIT rewards loops and recursion in your own code; it does little for code that mostly calls builtins.** If a profile shows time inside one builtin, the fix is to call it less often, not to look for a faster backend.

A tight arithmetic loop shows the same effect at its most extreme:

```
bind start = do {
  bind t0 = time_now()
  bind i = 0
  bind acc = 0
  while i < 3000000 {
    bind acc = acc + i % 7
    bind i = i + 1
  }
  println("acc = ", acc)
  println("seconds: ", time_now() - t0)
}
```

```
OUTPUT
acc = 8999994
seconds: 0.0031850337982177734
```

On the interpreter the same loop took 1.04 seconds. Running it with `--O0` took 0.0034 s, so for a loop this simple the MIR optimisation level barely matters. Across levels `fib(35)` took 37 ms at O0 and 31 ms at O3. Do not tune the `--O` flag until a profile shows a function where it changes the result; keep O3 and use `--O0` only as an escape hatch when compile time on a giant function becomes a problem.

A benchmarking method that holds up

1. **Time inside the program.** `time_now()` returns seconds since the program started, as a number with fractional digits. Subtract two readings and multiply by 1000 for milliseconds. Do not time the whole `ling run` process: on the machine used for this chapter process start-up dominated everything below a second.
2. **Make the work big enough.** If a measurement is a few milliseconds, scale the loop count until it is not. Compare ratios, not single numbers.
3. **Mind constant folding.** As the MIR above showed, O3 will fold a computation whose inputs are literals. Feed benchmarks values the optimiser cannot know (a loop counter, a value read from the environment or a file) and use the result, for example by printing it.
4. **Run each variant several times** and take the median. The first run can include disk cache effects.
5. **Compare backends explicitly** (`ling run` against `--interp`) and report both. A speed-up that exists only on one backend is not a property of your algorithm.
6. **Keep I/O out of the timed region** unless I/O is what you are measuring.
7. **Change one thing at a time.**

> TIP
>
> Give every benchmark function its own `time_now()` pair, as above. A `bench_*` function that prints name, result and milliseconds is easy to diff between commits.

Chapter summary

- A file is lexed, parsed to an AST, lowered to MIR, and then either JIT-compiled (default) or interpreted. The interpreter is the language; the JIT only accelerates your own arithmetic, loops and calls.
- `ling run` uses the JIT at O3 by default, the interpreter with `--interp`, and the interpreter automatically for programs containing `http_serve` or `lingtp_serve` and for every `thread_spawn` isolate.

- On divergence, the JIT stays quiet with `()` where the interpreter raises an error: unknown functions, `1/0`, bad indexes, missing files, closure arguments. It also truthy-fies empty text and lists, may mutate lists in place, and persists `bind` inside `match` arms.
- Correct-on-both code uses `while` / `if` /named functions, rebinds lists, guards inputs, compares text with `==`, and never passes functions to builtins. Test with a two-backend script.
- `ling visualize` draws a function-and-call overview; `ling ast` renders a project into `AST/`; `lingc mir -O n` shows what the optimiser did.
- Compiled code is hundreds of times faster on pure computation and only modestly faster on builtin-heavy code. Benchmark inside the program with `time_now()`, scale the work, and beware constant folding.

Exercises

1. Write a program in which `1 / 0` is guarded so that it behaves identically on both backends and returns a `bad(...)` result for a zero divisor. Confirm with `dualrun.sh`.
2. A teammate's program counts words with `for w in words { bind count = count + 1 }` and always prints 0. Explain why and fix it.
3. You must sum the numbers 0 to 9,999,999 of `i % 7`. A colleague proposes splitting the work across four isolates for speed. Using what this chapter showed about backends, say whether that is likely to beat a plain `while` loop in the main program, and why.

Answers

1. One solution:

```
fn safe_div(a, b) {
  if b == 0 { return bad("divisor is zero") }
  ok(a / b)
}

bind start = do {
  println(safe_div(10, 4))
  println(safe_div(1, 0))
}
```

```
OUTPUT
Ok(2.5)
Err(divisor is zero)
```

Both backends print the same text, so `dualrun.sh` reports `OK`.

2. A `for` body gets a fresh copy of the surrounding scope on every iteration, so `bind count = count + 1` updates a copy that is thrown away. This is a language rule and holds on both backends. Use `bind count = 0`, `bind i = 0` and `while i < len(words) { bind count = count + 1 bind i = i + 1 }`, or simply use `len(words)`.

3. Probably not. Isolates run on the interpreter, while the main program runs JIT-compiled. In the measurements for this book, summing `i % 7` over 8,000,000 values took about 14 ms in the main program and about 1.5 seconds in one isolate; four isolates brought that to about 0.5 seconds. Parallelism cannot recover a gap of two orders of magnitude. Isolates pay off for work that blocks (I/O, waiting) or that runs inside a builtin, not for arithmetic that the JIT already makes cheap. See *Concurrency and Isolates*.

CHAPTER 2

Concurrency and Isolates

What you'll learn

Ling has real parallelism, and it has several words that look like parallelism but are not. This chapter separates the two. You will learn how `thread_spawn` and `thread_join` work, what an isolate is and why it makes data races impossible, how to move data with channels, how capabilities limit what a worker may touch, and which patterns are safe. You will also see honest measurements: isolates are interpreters, so they are not automatically faster than your main program.

The chapter ends with a mini-project, a parallel file hasher built as a job queue with a pool of workers. It was run on the installed `ling` (2030.1.53) under both the default JIT and `--interp`, and the output below is identical on both unless the text says otherwise. The previous chapter, *How Ling Runs*, explains why the two backends matter here.

What `async` and `spawn` really do today

Start by clearing away the words that do nothing, because they are the first thing a programmer coming from another language will try.

```
async fn work(n) {
  sleep(200)
  println("working on ", n)
  n * 2
}

bind start = do {
  bind t0 = time_now()
  bind a = work(1)
  println("after first call (already finished): ", a)
  bind b = work(2)
  println("both done, elapsed >= 0.4s: ", time_now() - t0 >= 0.4)
}
```

```
OUTPUT
working on 1
after first call (already finished): 2
working on 2
both done, elapsed >= 0.4s: true
```

`async` is accepted and ignored. The function ran to completion, in order, before the next line executed; the two 200 ms sleeps added up to 400 ms. `wait` is just as hollow (it returns its value on the interpreter and `()` on the JIT). And `spawn` is a reserved word with no builtin behind it:

```
bind start = do {
  println(spawn(1))
}
```

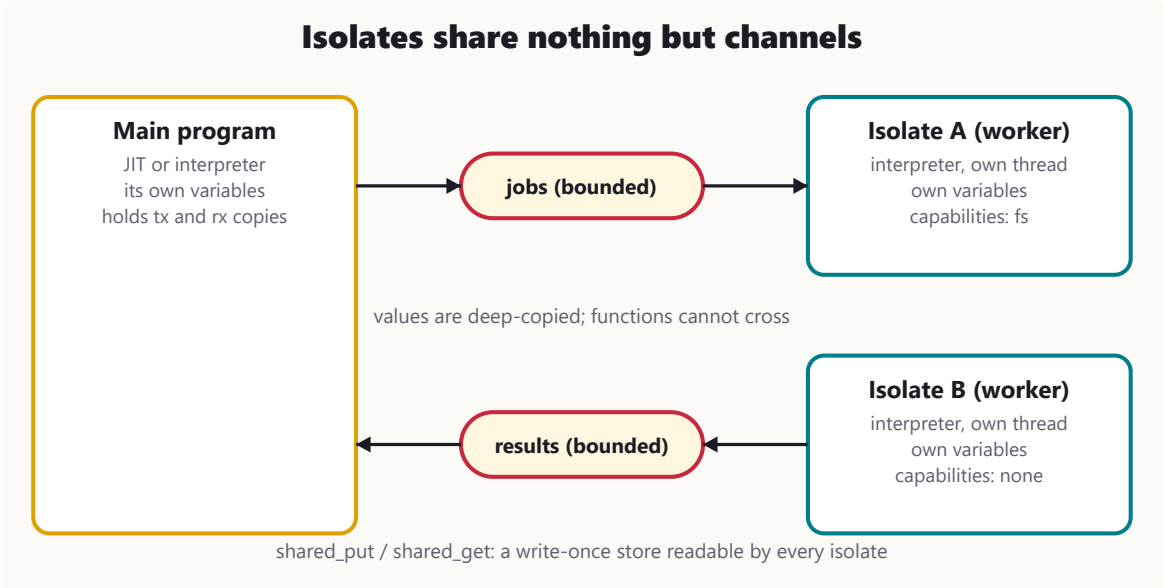
```
OUTPUT
()
```

On the JIT, an unknown function quietly yields `()`. On `--interp` the same file stops with `error[runtime]: unknown function 'spawn'`. Either way, nothing was started. All genuine concurrency in Ling goes through the `thread_*`, `chan_*` and `shared_*` builtins described below.

> **WARNING**
>
> Do not write `async fn` expecting overlap. It compiles, runs, gives the right answer and is completely sequential, which is the worst kind of wrong.

The isolate model

`thread_spawn(file, function, args, capabilities)` starts a function from another source file in a new **isolate**: a fresh interpreter on its own operating-system thread. Nothing is shared between isolates except channels and a write-once store. Because no mutable state is ever reachable from two threads, Ling programs cannot have data races, and there are no locks to forget.



Isolates exchange deep copies of values through bounded channels. Handles are unguessable strings that only work in the isolate that owns them.

The rules, with the limits from the runtime:

- Arguments, messages and return values are **deep-copied**. Functions cannot be sent.
- Isolates **always run on the interpreter**, even when the parent is JIT-compiled.
- Handles (thread handles and channel ends) are 128-bit random hex strings, valid only in the isolate that owns or was given them.
- At most 1024 isolates may be live at once; a message may be at most 64 MiB; a channel holds 1 to 65,536 queued messages.
- A worker's function lives in **another file**, named relative to the calling file. A function cannot spawn itself.

Here is the worker file used for the rest of the chapter. Save it as `worker.ling` next to your programs. It has no entry point; it is just a library of functions that isolates can run.

FRAGMENT

```
fn square(n) { n * n }

fn sum_range(lo, hi) {
  bind i = lo
  bind acc = 0
  while i < hi {
    bind acc = acc + i % 7
    bind i = i + 1
  }
  acc
}

fn boom(x) { no_such_builtin_xyz(x) + 1 }

fn try_read(path) { read_file(path) }

fn spin(n) {
  bind i = 0
  while i < n { bind i = i + 1 }
  i
}

fn echo(x) { x }

fn read_cfg() { shared_get("cfg", "none") }

fn slow_send(tx, label, ms) {
  sleep(ms)
  chan_send(tx, label)
  label
}

bind counter = 100
fn see_global() { counter }
```

Your first thread

`thread_spawn` returns a result, `ok(handle)` or `bad(reason)`. The handle is wrapped, so unwrap it before passing it to `thread_join`. Every program in this chapter uses this one-line helper:

```
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

bind start = do {
  bind h = payload(thread_spawn("worker.ling", "square", [12], []))
  println("handle is text of length ", len(h))
  println("joined: ", thread_join(h))
  println("alive after join: ", thread_alive(h))
}
```

```
OUTPUT
handle is text of length 32
joined: Ok(144)
alive after join: false
```

`thread_join` waits and returns `ok(value)` with the function's return value. If the worker raised an error, panicked or was cancelled you get `bad(...)` instead, and the error never crashes your process. That makes join the natural place to collect failures.

Failures, capabilities and missing files

```
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

bind start = do {
  bind h1 = payload(thread_spawn("worker.ling", "boom", [1], []))
  println("runtime error : ", thread_join(h1))

  write_file("data.txt", "secret file\n")
  bind h2 = payload(thread_spawn("worker.ling", "try_read", ["data.txt"], []))
  println("no capability : ", thread_join(h2))

  bind h3 = payload(thread_spawn("worker.ling", "try_read", ["data.txt"], ["fs"]))
  println("with fs       : ", thread_join(h3))

  println("missing file  : ", thread_spawn("nofile.ling", "x", [], []))
  remove_file("data.txt")
}
```

```
OUTPUT
runtime error : Err(unknown function 'no_such_builtin_xyz')
no capability : Err(capability "fs" not granted)
with fs       : Ok(secret file
)
missing file  : Err(cannot find 'nofile.ling')
```

Four things are worth noticing.

1. Inside an isolate a call to an unknown function is an **error**, not a silent `()`, because isolates run on the interpreter. A bug the JIT would hide in the main program is caught in a worker.
2. The fourth argument to `thread_spawn` is a list of **capabilities**: any of `fs`, `db`, `net`, `p2p`, `proc`, `keys`, `clipboard`. A worker gets none by default, and a builtin that needs a missing capability fails with `capability "fs" not granted`. A child can never be given a capability its parent lacks. `thread_caps()` in the main program lists all seven.
3. Spawning a file that does not exist is an ordinary `bad(...)` from `thread_spawn` itself, not an error in the child.
4. Window, audio and other GUI builtins are not available in isolates at all.

> **TIP**
>
> Treat capabilities as the least-privilege setting for each worker. A hashing worker gets `["fs"]`; a worker that parses untrusted text gets `[]`. If it is ever compromised by bad input, it still cannot read your files or open a socket.

Cancellation and timeouts

Cancellation is cooperative: the target aborts at its next loop iteration or call. `thread_join_timeout` lets you wait without committing.

```
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

bind start = do {
  bind h = payload(thread_spawn("worker.ling", "spin", [100000000000], []))
  sleep(50)
  println("alive:   ", thread_alive(h))
  println("timeout: ", thread_join_timeout(h, 100))
  println("cancel:  ", thread_cancel(h))
  println("join:    ", thread_join(h))
  println("alive:   ", thread_alive(h))
}
```

```
OUTPUT
alive:   true
timeout: Err(timeout)
cancel:  true
join:    Err(cancelled)
alive:   false
```

The worker was spinning through a hundred billion iterations. A timeout leaves it running; `thread_cancel` returns `true` because it was still running and not yet cancelled; the join then reports `Err(cancelled)`.

Nothing is shared

Now the property that gives the model its safety, shown by trying to break it.

```
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

bind counter = 1

bind start = do {
  // 1. Isolates do not share variables.
  bind h = payload(thread_spawn("worker.ling", "see_global", [], []))
  println("child sees counter = ", thread_join(h), ", parent has ", counter)

  // 2. The write-once shared store.
  println(shared_put("cfg", [1, 2, 3]))
  bind h2 = payload(thread_spawn("worker.ling", "read_cfg", [], []))
  println("child reads: ", thread_join(h2))
  println("second put:  ", shared_put("cfg", [9]))
  println("missing key: ", shared_get("nope", "none"))

  // 3. Values are deep-copied across the boundary.
  bind data = [1, 2, 3]
  bind h3 = payload(thread_spawn("worker.ling", "echo", [data], []))
  bind back = payload(thread_join(h3))
  println("echoed: ", back)

  // 4. Functions cannot be sent.
  println(match thread_spawn("worker.ling", "echo", [(|x| x + 1)], []) { ok(h) =>
"spawned", bad(e) => e })

  // 5. chan_select: first channel to deliver wins.
  bind fast = chan_new(1)
  bind slow = chan_new(1)
  bind a = payload(thread_spawn("worker.ling", "slow_send", [slow[0], "slow", 300],
[]))
  bind b = payload(thread_spawn("worker.ling", "slow_send", [fast[0], "fast", 20], []))
  println("select: ", chan_select([slow[1], fast[1]], 2000))
  println("select: ", chan_select([slow[1], fast[1]], 2000))
  println("select (nothing left): ", chan_select([slow[1], fast[1]], 50))
}
```

```
OUTPUT
child sees counter = Ok(100), parent has 1
Ok(())
child reads: Ok([1, 2, 3])
second put:  Err(exists)
missing key: none
echoed: [1, 2, 3]
spawned
select: Ok([1, fast])
select: Ok([0, slow])
select (nothing left): Err(timeout)
```

Read it step by step.

1. The worker file's own top-level `counter` is 100 and the parent's is 1. They are different variables in different interpreters.
2. `shared_put` is a process-wide store whose keys can never be overwritten. It is the right place for configuration every worker needs; it can never leak a handle, because handles are not translated into the store.
3. Handing a list to a worker gives it a copy. Changes the worker makes to its copy never reach yours, and vice versa.
4. Step 4 shows a JIT/interpreter divergence. On the JIT, the closure argument is lowered to `()`, so the spawn *succeeds* and the worker would receive `()`. With `--interp` the same line prints `functions cannot be sent between threads`. Do not rely on either; never pass a function across the boundary, send data and let the worker call a function by name.
5. `chan_select(handles, timeout_ms)` returns `ok([index, value])` for the first channel that delivers: the 20 ms sender (index 1) beat the 300 ms sender (index 0). It returns `bad("timeout")` if nothing arrives in time and `bad("closed")` if every channel is closed.

Channels: pipelines and the closing rules

`chan_new(capacity)` returns a two-element list `[send_handle, receive_handle]`. Channels are bounded, multi-producer multi-consumer queues; a full channel blocks the sender (`chan_send`), an empty one blocks the receiver (`chan_recv`). Non-blocking forms exist: `chan_try_send` and `chan_try_recv` return `bad("full")` and `bad("empty")`, and `chan_recv_timeout(h, ms)` returns `bad("timeout")`. When the channel is closed and drained, `chan_recv` returns `bad("closed")`.

A two-stage pipeline shows the typical shape. The workers go in `pipe.ling`:

```
FRAGMENT
fn is_ok(r) { match r { ok(v) => true, bad(e) => false } }
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

fn producer(tx, count) {
  for i in 0..count { chan_send(tx, i * 10) }
  chan_close(tx)
  count
}

fn square_stage(rx, tx) {
  bind going = true
  bind n = 0
  while going {
    bind m = chan_recv(rx)
    bind going = is_ok(m)
    if going {
      chan_send(tx, payload(m) * payload(m))
      bind n = n + 1
    }
  }
  chan_close(tx)
  n
}
```

and the main program wires them together:

```
fn is_ok(r) { match r { ok(v) => true, bad(e) => false } }
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

bind start = do {
  bind a = chan_new(4)
  bind b = chan_new(4)
  bind p = payload(thread_spawn("pipe.ling", "producer", [a[0], 6], []))
  bind s = payload(thread_spawn("pipe.ling", "square_stage", [a[1], b[0]], []))
  bind going = true
  bind got = []
  while going {
    bind m = chan_recv(b[1])
    bind going = is_ok(m)
    if going { bind got = list_push(got, payload(m)) }
  }
  println("received: ", got)
  println("producer returned: ", thread_join(p))
  println("stage returned:    ", thread_join(s))
}
```

```
OUTPUT
received: [0, 100, 400, 900, 1600, 2500]
producer returned: Ok(6)
stage returned:    Ok(6)
```

Values flowed through two bounded channels of capacity four while the producer sent six, so back-pressure was exercised too. Two details make this loop work, and both are the source of the most common concurrency bugs in Ling.

Why the loops use `while` plus a helper, not `match` with `bind` inside. The natural way to write the receive loop is `match chan_recv(rx) { ok(v) => {...}, bad(e) => { bind going = false } }`. A `bind` inside a `match` arm persists on the JIT but is discarded on the interpreter. Isolate code always runs on the interpreter, so `going` never becomes false and the worker spins forever. Compute the flag with a `match` *expression* (`is_ok` above), rebind it outside, and the loop behaves the same everywhere.

Why the producer closes explicitly. Ending a channel is *not* automatic when a thread exits if someone else still holds the send end. The main program here received a copy of `a[0]` when it created the channel, so closing is the producer's explicit job:

```
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

bind start = do {
  bind ch = chan_new(4)
  bind h = payload(thread_spawn("w3.ling", "two_then_exit", [ch[0]], []))
  println(thread_join(h))
  println(chan_recv(ch[1]))
  println(chan_recv(ch[1]))
  println("third recv: ", chan_recv_timeout(ch[1], 200))
  chan_close(ch[0])
  println("after close: ", chan_recv_timeout(ch[1], 200))
}
```

OUTPUT

```
Ok(producer exited without closing)
Ok(one)
Ok(two)
third recv: Err(timeout)
after close: Err(closed)
```

with `w3.ling`:

FRAGMENT

```
fn two_then_exit(tx) {
  chan_send(tx, "one")
  chan_send(tx, "two")
  "producer exited without closing"
}
```

The producer returned, yet the channel stayed open (`timeout`, not `closed`) because the main program still held a send handle. `chan_close(handle)` closes that side of the channel for everybody; messages already queued are still delivered, and the receiver then sees `closed`. A plain `chan_recv` in that situation would have blocked forever, which is exactly how a missing `chan_close` shows up in practice: a hang, not an error.

> **WARNING**
>
> A blocking receive on a channel nobody will ever close or send to never returns. Use `chan_recv_timeout` or `chan_select` with a timeout in any code where a worker might die early, and always close the send end when a producer is done.

> **NOTE**
>
> `chan_new_secure(capacity)` behaves the same but zeroises string payloads after delivery. Use it when messages carry secrets.

What parallelism buys you

Since isolates are interpreters, the speed-up arithmetic needs care. This program sums `i % 7` for the first eight million integers three ways: in the main program, in one isolate, and split across four isolates.

```
use "worker.ling"

fn payload(r) { match r { ok(v) => v, bad(e) => e } }

fn ms_since(t0) { round((time_now() - t0) * 1000) }

fn run_isolates(n, total) {
  bind span = total / n
  bind hs = []
  bind k = 0
  while k < n {
    bind h = payload(thread_spawn("worker.ling", "sum_range", [k * span, (k + 1) *
span], []))
    bind hs = list_push(hs, h)
    bind k = k + 1
  }
  bind acc = 0
  bind j = 0
  while j < n {
    bind acc = acc + payload(thread_join(hs[j]))
    bind j = j + 1
  }
  acc
}

bind start = do {
  bind total = 8000000

  bind t0 = time_now()
  bind r1 = sum_range(0, total)
  println("in-process (", "main program", "): ", r1, "  ", ms_since(t0), " ms")

  bind t1 = time_now()
  bind r2 = run_isolates(1, total)
  println("1 isolate:           ", r2, "  ", ms_since(t1), " ms")

  bind t2 = time_now()
```

```
  bind r3 = run_isolates(4, total)
  println("4 isolates:          ", r3, "  ", ms_since(t2), " ms")
}
```

```
OUTPUT
in-process (main program): 23999997  15 ms
1 isolate:           23999997  1461 ms
4 isolates:          23999997  498 ms
```

Under `--interp` the first line is the slow one (1540 ms) and the three agree to within noise:

```
OUTPUT
in-process (main program): 23999997  1540 ms
1 isolate:           23999997  1472 ms
4 isolates:          23999997  456 ms
```

The result is identical everywhere (a handy check that you split the range correctly), and the lesson is blunt. Four isolates run about three times faster than one, which is good scaling. But the JIT-compiled main program was roughly a hundred times faster than the four-isolate version. **Do not reach for threads to speed up arithmetic the JIT already handles.** Reach for them when:

- the work **blocks**: waiting on sockets, sleeping, polling, slow subprocesses;
- the work is **inside a builtin** that releases the main program's time, such as hashing files, compressing, or database calls (each isolate gets its own capability-gated access);
- you need **isolation** for its own sake: a worker that parses hostile input, or a long job that must be cancellable;
- the work is **heavy enough per message** that the copy and the interpreter cost are small next to it.

> NOTE
>
> `thread_cpus()` reports how many hardware threads exist. Size a pool from it rather than hard-coding a number, and remember a pool larger than your cores buys nothing for CPU-bound work.

Safe patterns

These rules came out of the experiments above.

1. **Fan-out, fan-in with indexes.** Number the jobs, have workers reply `[index, result]`, and place results by index so output order never depends on scheduling.
2. **One-way data flow.** Give each channel one producer role and one consumer role. A job channel and a result channel is easier to reason about than a shared bidirectional one.
3. **Size result channels for the job count**, or interleave sending and receiving. If the main program blocks sending jobs while workers block sending results into a full channel, everything deadlocks.
4. **Explicit close, bounded waits.** Close the send end when you are done. Use `thread_join_timeout`, `chan_recv_timeout` and `chan_select` timeouts on every path that could stall.
5. **Least privilege.** Pass the smallest capability list that works.
6. **No rebinding inside `match` arms** in any code that may run on both backends, and always in worker code.
7. **Send data, not behaviour.** Pass names or numbers that select a function inside the worker file.
8. **Treat `bad(...)` from `thread_join` as a normal outcome.** A worker that fails should degrade one job, not your program.

> TRY IT
>
> Change `run_isolates(4, total)` to use `thread_cpus()` and print how long it takes for 1, 2, 4 and 8 isolates. Where does the curve flatten on your machine?

Mini-project: a parallel file hasher

The task: hash many files with a pool of workers that pull paths from a shared job queue, and report them in the original order. It uses every piece above: a job channel closed to signal "no more work", a result channel, the `fs` capability, indexed replies, and a serial cross-check.

The worker, in `hasher_worker.ling`:

```
FRAGMENT
fn is_ok(r) { match r { ok(v) => true, bad(e) => false } }
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

// A job is [index, path]. The reply is [index, hash], so the caller can
// restore the original order no matter which worker finishes first.
fn hash_worker(jobs, results) {
  bind going = true
  bind done = 0
  while going {
    bind m = chan_recv(jobs)
    bind going = is_ok(m)
    if going {
      bind job = payload(m)
      chan_send(results, [job[0], file_hash(job[1])])
      bind done = done + 1
    }
  }
  done
}
```

The driver creates a fixture, starts four workers, queues the jobs, closes the queue and collects:

```
fn is_ok(r) { match r { ok(v) => true, bad(e) => false } }
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

fn make_fixture(count) {
  bind paths = []
  bind i = 0
  while i < count {
    bind p = "fixture/file" + str(i) + ".txt"
    write_file(p, "contents of file number " + str(i) + "\n")
    bind paths = list_push(paths, p)
    bind i = i + 1
  }
  paths
}

fn start_workers(n, jobs, results) {
  bind hs = []
  bind w = 0
  while w < n {
    bind h = payload(thread_spawn("hasher_worker.ling", "hash_worker", [jobs[1],
results[0]], ["fs"]))
    bind hs = list_push(hs, h)
    bind w = w + 1
  }
  hs
}

fn collect(results, n) {
  bind hashes = []
  bind i = 0
  while i < n {
    bind hashes = list_push(hashes, "")
    bind i = i + 1
  }
  bind k = 0
  while k < n {
    bind reply = payload(chan_recv(results[1]))
    bind hashes = list_set(hashes, reply[0], reply[1])
```

```
    bind k = k + 1
  }
  hashes
}

bind start = do {
  bind paths = make_fixture(12)
  bind n_files = len(paths)
  bind jobs = chan_new(n_files)
  bind results = chan_new(n_files)
  bind workers = start_workers(4, jobs, results)

  bind j = 0
  while j < n_files {
    chan_send(jobs[0], [j, paths[j]])
    bind j = j + 1
  }
  chan_close(jobs[0])

  bind hashes = collect(results, n_files)
  bind shown = 0
  while shown < 3 {
    println(paths[shown], "  ", substr(hashes[shown], 0, 16), "...")
    bind shown = shown + 1
  }

  bind handled = 0
  bind w = 0
  while w < len(workers) {
    bind handled = handled + payload(thread_join(workers[w]))
    bind w = w + 1
  }
  println("files hashed by workers: ", handled)

  bind bad_count = 0
  bind m = 0
  while m < n_files {
    if file_hash(paths[m]) != hashes[m] { bind bad_count = bad_count + 1 }
    bind m = m + 1
```

```
    }
    println("mismatches vs serial: ", bad_count)
    remove_dir_all("fixture")
}
```

```
OUTPUT
fixture/file0.txt  9d872d6b81cb74e4...
fixture/file1.txt  e17818079c74b5ae...
fixture/file2.txt  8004a12e6e48d1d6...
files hashed by workers: 12
mismatches vs serial: 0
```

The design carries the lessons of the chapter. Closing `jobs[0]` after queueing is the "no more work" signal: workers drain the queue and then see `closed`, which ends their loop and makes `thread_join` return their job counts, summing to 12. Both channels are sized to the number of files, so neither the producer nor the workers can block each other. The indexed replies mean the printed order is stable on every run even though the workers finish in any order. And `file_hash` is a native builtin, so the hash equals what the main program computes serially, which the final check confirms.

One honest measurement. With 16 files of about 10 MB each (made with `str_repeat`), four workers finished in about 200 ms and the serial check pass took about 300 ms. That is a modest 1.5 times, not 4, because hashing is memory- and disk-bound and the hash itself already runs at native speed in both cases. The pattern is worth having; the speed-up on this kind of job is real but not dramatic.

> TIP
>
> Production variations of the same skeleton: add a third channel for progress messages that the main program polls with `chan_try_recv`; give each job a retry count and requeue failures; replace `file_hash` with any capability-gated builtin (`read_file`, `db_query`, a network call with the `net` capability).

Chapter summary

- `async`, `wait` and `spawn` do not create concurrency: `async fn` runs synchronously, `wait` is unreliable, and `spawn` is not implemented. The real tools are `thread_spawn/join`, `chan_*` and `shared_*`.

- Each thread is an isolate: its own interpreter, no shared variables, deep-copied messages, no functions across the boundary. Data races cannot happen.
- `thread_spawn` returns `ok(handle)`; unwrap it. `thread_join` returns `ok(value)` or `bad(error)`, and worker errors never crash the parent. Cancellation is cooperative.
- Capabilities (`fs db net p2p proc keys clipboard`) are granted per spawn and never exceed the parent's.
- Channels are bounded. Close the send end explicitly: a thread exiting does not close a channel someone else still holds. Prefer bounded waits.
- Isolates are interpreters. Four isolates scale about 3x over one, but a JIT-compiled main program can beat them on pure arithmetic by two orders of magnitude. Use threads for blocking, I/O and isolation.
- Never rebind variables inside `match` arms in worker code; compute flags with `match` expressions.

Exercises

1. Write a worker `fn count_lines(path)` that returns the number of lines in a text file, and a main program that spawns it with the `fs` capability on a file you create, printing the joined result. Then remove the capability and show what happens.
2. Modify the pipeline so the main program receives from two producers using `chan_select`, printing each value with its source index, until both channels are closed. State what `chan_select` returns when the last channel closes.
3. A colleague's worker loop reads `match chan_recv(rx) { ok(v) => { bind total = total + v } bad(e) => { bind going = false } }` and hangs when run under `thread_spawn` although it works when called directly in `start`. Explain why, and rewrite it.

Answers

1. The worker:

```
FRAGMENT
fn count_lines(path) { len(split(read_file(path), "\n")) }
```

The driver:

```
fn payload(r) { match r { ok(v) => v, bad(e) => e } }

bind start = do {
  write_file("three.txt", "a\nb\nc")
  bind h = payload(thread_spawn("lines_worker.ling", "count_lines", ["three.txt"],
["fs"]))
  println(thread_join(h))
  bind h2 = payload(thread_spawn("lines_worker.ling", "count_lines", ["three.txt"],
[]))
  println(thread_join(h2))
  remove_file("three.txt")
}
```

OUTPUT

```
Ok(3)
Err(capability "fs" not granted)
```

(Save the worker as `lines_worker.ling`.) Note that `split` on a text without a trailing newline gives exactly three elements.

2. Use two channels, each with its own producer that closes its own send end, and loop on `chan_select`. The producer, in `sel_w.ling`:

FRAGMENT

```
fn feed(tx, label, n) {
  for i in 0..n { chan_send(tx, label + str(i)) }
  chan_close(tx)
  n
}
```

The driver:

```
fn payload(r) { match r { ok(v) => v, bad(e) => e } }
fn is_ok(r) { match r { ok(v) => true, bad(e) => false } }

bind start = do {
  bind a = chan_new(4)
  bind b = chan_new(4)
  bind pa = payload(thread_spawn("sel_w.ling", "feed", [a[0], "a", 2], []))
  bind pb = payload(thread_spawn("sel_w.ling", "feed", [b[0], "b", 3], []))
  bind going = true
  while going {
    bind r = chan_select([a[1], b[1]], 1000)
    bind going = is_ok(r)
    if going { println("source ", payload(r)[0], " -> ", payload(r)[1]) } else {
println("select ended: ", r) }
  }
}
```

```
OUTPUT
source 0 -> a0
source 0 -> a1
source 1 -> b0
source 1 -> b1
source 1 -> b2
select ended: Err(closed)
```

When every channel in the list is closed and drained, `chan_select` returns `bad("closed")`, which ends the loop. (The interleaving of sources can vary between runs; here the 2 and 3 messages happened to arrive in this order on both backends.) `bad("timeout")` would mean nothing arrived in the time limit while some channel is still open.

3. The worker runs on the interpreter, where a `bind` inside a `match` arm is discarded. `going` therefore never becomes `false` and `total` never accumulates, so the loop spins on the closed channel forever. Called directly in `start` under the default JIT, binds in arms persist, which hides the bug. The portable rewrite computes the flag with a `match` expression in a helper (`is_ok`), rebinds it outside the match, and accumulates inside an `if`, exactly as in `square_stage` above.

CHAPTER 3

Network Services in Ling

What you'll learn

By the end of this chapter you will have written, run and attacked a real network service: a JSON-over-HTTP notes API with SQLite storage, token authentication, input validation, a login throttle and persistence across restarts. You will also write a client for it. More importantly, you will know how the Ling HTTP runtime behaves under load and under failure, because that behaviour, not the syntax, is what separates a demo from something you can leave running.

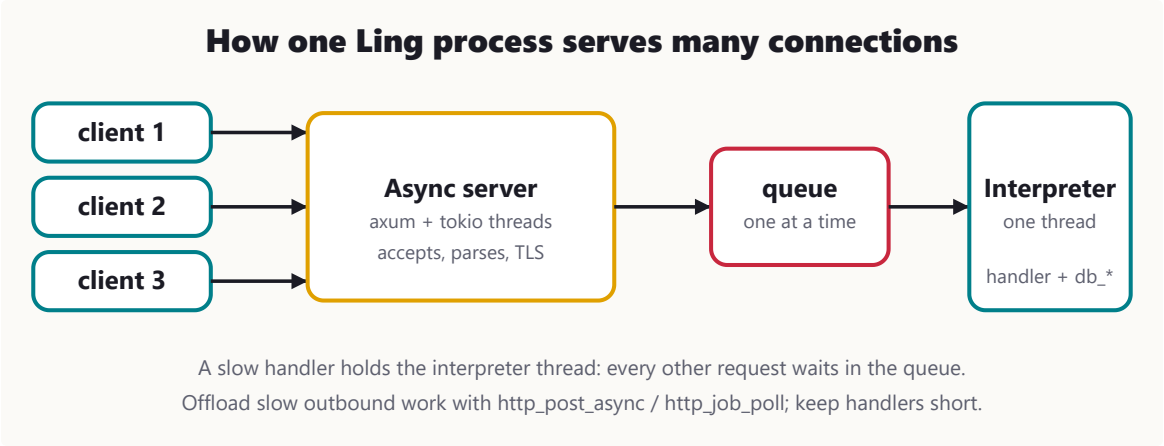
Specifically, you will learn:

- how the Ling server dispatches requests, and what a single-threaded handler loop means for design;
- how to structure handlers, validation and error responses so that every failure has a status code and a safe body;
- how to authenticate clients with Argon2id password hashes and server-side session tokens;
- what the runtime does *not* do for you (CORS, security headers, client addresses, access logs), with the measured evidence;
- how to write a client with `http_get` and `http_post_async`, and where its limits are;
- a deployment and hardening checklist built from things that actually went wrong while testing.

Every program in this chapter was run against `ling` 2030.1.53 on loopback, and every transcript is real. The reference chapter *Networking and the Web* lists each builtin; this chapter is about putting them together. You should be comfortable with `form`, `while` loops and the interpreter's scoping rules from the earlier tutorials. If `bind` inside a `for` body not updating an outer variable surprises you, revisit *Forms and Choices* first, because the services below depend on that rule.

How the server really works

The first thing to internalise is the request path. The HTTP listener is an `axum` server on a pool of background threads. It accepts connections, terminates TLS if you asked for it, parses the request and then hands it to *your* handler. But Ling values live on one thread, so every handler call is funnelled onto the interpreter thread, one request at a time.



Many clients may be connected at once, but handlers run strictly one after another on the interpreter thread.

That design has a clean benefit: your handlers never race. There are no locks, and the one SQLite connection that `db_open` creates needs none. The cost is that a slow handler stalls everybody. Prove it to yourself before you design around it:

```
GUI
fn boom(req) {
  db_exec("INSERT INTO missing_table VALUES (1)")
  return "unreachable"
}

fn slow(req) {
  sleep(1000)
  return "slow done"
}

fn whoami(req) { return "ip=[" + req.client_ip + "]" }

fn quick(req) { return "quick" }

bind start = do {
  db_open(":memory:")
  http_route("GET", "/boom", boom)
  http_route("GET", "/slow", slow)
  http_route("GET", "/quick", quick)
  http_route("GET", "/ip", whoami)
  http_serve("127.0.0.1", 18111)
}
```

Start it, then fire two `/slow` requests and one `/quick` request at the same instant. The quick request has to wait its turn:

```
OUTPUT
$ curl -s $H/slow & curl -s $H/slow & curl -s -w 'quick answered at %{time_total}s\n'
$H/quick
quick answered at 1.960388s
```

A handler that does nothing took nearly two seconds because two one-second handlers were ahead of it in the queue. The same program shows two more behaviours you must design around.

```
OUTPUT
$ curl -s -i $H/boom
HTTP/1.1 500 Internal Server Error
handler error: Runtime("db_exec: no such table: missing_table\n  sql: INSERT INTO
missing_table VALUES (1)")
$ curl -s $H/ip
ip=[]
$ curl -s -H 'X-Forwarded-For: 6.6.6.6, 10.0.0.1' $H/ip
ip=[6.6.6.6]
```

Read those two transcripts slowly. First, an error that escapes a handler becomes a `500` whose body contains the interpreter's error text, including your SQL. That is an information leak. Second, `client_ip` is empty for a direct connection, and when it is set, it is whatever the *client* put in `X-Forwarded-For`. It is only trustworthy behind a proxy that overwrites the header.

> **WARNING**
>
> A `500` body in this runtime is `handler error:` followed by the raw internal error. Never let an error escape a handler in a public service. The service below validates first and, in the hardening section, wraps handlers in `try`.

> **NOTE**
>
> `ling run` silently selects the interpreter for any file that calls `http_serve`, because of a known closure-argument bug in the JIT. Your server therefore runs interpreted whatever you pass on the command line, so testing with `--interp` changes nothing. Handlers are I/O bound anyway; the JIT would rarely matter.

Designing the API before the code

Ling's router matches the HTTP method and the *exact* path. There are no path parameters, so identifiers travel in the query string: `/note?id=7`, not `/notes/7`. Decide your resource shape with that in mind. Ours is:

Method and path	Auth	Purpose	Failure statuses
`GET /health`	none	Liveness probe	none
`POST /login`	none	Exchange name and password for a token	401, 429

Method and path	Auth	Purpose	Failure statuses
`GET /notes?limit=N`	token	List the caller's notes, newest last	401
`POST /notes`	token	Create a note	400, 401, 413
`GET /note?id=N`	token	Fetch one of the caller's notes	401, 404
`DELETE /note?id=N`	token	Delete one of the caller's notes	401, 404

Two decisions deserve a sentence each. **Input is form-encoded**, because Ling has no JSON parser: `query_param(req.body, "title", "")` parses `k=v&k=v` and percent-decodes for you. **Output is JSON assembled by hand**, with every string passing through `json_escape`. If you find that distasteful, that is a fair signal that a service with rich JSON input belongs in Rust (see *Web Apps with the Rust Crates*).

Authorisation is part of the query, not an afterthought: every `SELECT` and `DELETE` carries `AND owner = ?`. A note belonging to someone else simply does not exist for you, and the API answers `404`, never `403`, so it does not confirm that an id is in use.

The service

Here is the whole thing. Read it once through, then we will walk the important parts.

```
GUI
form Response { status, content_type, body }

fn json(status, body) {
  return Response(status, "application/json", body)
}

fn fail(status, msg) {
  println("warn status=", status, " ", msg)
  return json(status, "{\"error\":\"" + json_escape(msg) + "\"}")
}

fn note_json(r) {
  return "{\"id\":" + r.id + ",\"title\":\"" + json_escape(r.title) + "\",\"body\":\""
+ json_escape(r.body) + "\"}"
}

// Returns the user name for a valid token, or "" if none. The token comes
// from "Authorization: Bearer <token>" or, for clients that cannot set
// headers (the Ling http_get), from ?token=.
fn current_user(req) {
  bind token = query_param(req.query, "token", "")
  if starts_with(req.authorization, "Bearer ") {
    bind token = substr(req.authorization, 7, len(req.authorization) - 7)
  }
  if token == "" { return "" }
  bind rows = db_query("SELECT user FROM sessions WHERE token = ?1 AND expires > ?2",
token, epoch_now())
  if len(rows) == 0 { return "" }
  return rows[0].user
}

fn health(req) {
  return json(200, "{\"status\":\"ok\"}")
}

fn login(req) {
  bind user = query_param(req.body, "user", "")
```

```
  bind pw = query_param(req.body, "password", "")
  bind since = epoch_now() - 300
  bind recent = db_query("SELECT count(*) AS n FROM attempts WHERE user = ?1 AND ts > ?
2", user, since)
  if recent[0].n >= 5 {
    return fail(429, "too many attempts, try again later")
  }
  bind rows = db_query("SELECT hash FROM users WHERE name = ?1", user)
  if len(rows) == 0 || !password_verify(pw, rows[0].hash) {
    db_exec("INSERT INTO attempts(user, ts) VALUES (?1, ?2)", user, epoch_now())
    return fail(401, "bad credentials")
  }
  bind token = random_hex(16)
  db_exec("INSERT INTO sessions(token, user, expires) VALUES (?1, ?2, ?3)", token,
user, epoch_now() + 3600)
  println("info login user=", user)
  return json(200, "{\"token\":\"" + token + "\",\"expires_in\":3600}")
}

fn list_notes(req) {
  bind who = current_user(req)
  if who == "" { return fail(401, "login required") }
  bind limit = clamp(to_number(query_param(req.query, "limit", "20")), 1, 100)
  bind rows = db_query("SELECT id, title, body FROM notes WHERE owner = ?1 ORDER BY id
LIMIT ?2", who, limit)
  bind out = "["
  bind i = 0
  while i < len(rows) {
    if i > 0 { bind out = out + "," }
    bind out = out + note_json(rows[i])
    bind i = i + 1
  }
  return json(200, out + "]")
}

fn create_note(req) {
  bind who = current_user(req)
  if who == "" { return fail(401, "login required") }
  bind title = trim(query_param(req.body, "title", ""))
```

```
  bind body = query_param(req.body, "body", "")
  if title == "" { return fail(400, "title is required") }
  if len(title) > 80 { return fail(400, "title too long (max 80)") }
  if len(body) > 2000 { return fail(413, "body too large (max 2000)") }
  db_exec("INSERT INTO notes(owner, title, body) VALUES (?1, ?2, ?3)", who, title,
body)
  bind rows = db_query("SELECT id, title, body FROM notes WHERE owner = ?1 ORDER BY id
DESC LIMIT 1", who)
  println("info create user=", who, " id=", rows[0].id)
  return json(201, note_json(rows[0]))
}

fn show_note(req) {
  bind who = current_user(req)
  if who == "" { return fail(401, "login required") }
  bind id = to_number(query_param(req.query, "id", "0"))
  bind rows = db_query("SELECT id, title, body FROM notes WHERE id = ?1 AND owner = ?
2", id, who)
  if len(rows) == 0 { return fail(404, "not found") }
  return json(200, note_json(rows[0]))
}

fn delete_note(req) {
  bind who = current_user(req)
  if who == "" { return fail(401, "login required") }
  bind id = to_number(query_param(req.query, "id", "0"))
  bind n = db_exec("DELETE FROM notes WHERE id = ?1 AND owner = ?2", id, who)
  if n == 0 { return fail(404, "not found") }
  return json(200, "{\"deleted\":" + id + "}")
}

bind start = do {
  db_open(cli_arg("db", ":memory:"))
  db_exec("CREATE TABLE IF NOT EXISTS users(name TEXT PRIMARY KEY, hash TEXT NOT
NULL)")
  db_exec("CREATE TABLE IF NOT EXISTS sessions(token TEXT PRIMARY KEY, user TEXT NOT
NULL, expires REAL NOT NULL)")
  db_exec("CREATE TABLE IF NOT EXISTS attempts(user TEXT NOT NULL, ts REAL NOT NULL)")
  db_exec("CREATE TABLE IF NOT EXISTS notes(id INTEGER PRIMARY KEY AUTOINCREMENT, owner
```

```
TEXT NOT NULL, title TEXT NOT NULL, body TEXT NOT NULL DEFAULT '')")
  db_exec("INSERT OR IGNORE INTO users(name, hash) VALUES (?1, ?2)", "ada",
password_hash(cli_arg("ada_pw", "lovelace-1815")))
  db_exec("INSERT OR IGNORE INTO users(name, hash) VALUES (?1, ?2)", "bob",
password_hash("bob-pass-42"))
  http_route("GET", "/health", health)
  http_route("POST", "/login", login)
  http_route("GET", "/notes", list_notes)
  http_route("POST", "/notes", create_note)
  http_route("GET", "/note", show_note)
  http_route("DELETE", "/note", delete_note)
  println("listening")
  http_serve("127.0.0.1", to_number(cli_arg("port", "18110")))
}
```

Run it with `ling run svc.ling --port 18110`. Without `--db` it uses an in-memory database, which is what you want for experiments. The first thing it prints is `listening`, followed by the runtime's banner.

Handlers return values, and `fail` is the only way out

The `Response` form is declared with three fields, and the name `Response` does not matter; the *field names* `status`, `content_type` and `body` do. Every handler returns either a `Response` built by `json(...)` or by `fail(...)`. `fail` does two jobs: it logs a line starting with `warn` so that you can grep your logs for refused requests, and it builds the error body. Centralising this means no handler can forget the JSON error shape, and nothing but a hand-chosen message ever reaches the client.

Guards are plain functions

Ling gives you no middleware and, as the closure notes in the reference explain, you should not try to build one by passing handlers around as values. The reliable pattern is a plain guard function called at the top of every handler:

```
FRAGMENT
fn list_notes(req) {
  bind who = current_user(req)
  if who == "" { return fail(401, "login required") }
  ...
}
```

It is repetitive, but it is obvious, greppable and impossible to get subtly wrong. `current_user` accepts the token from `Authorization: Bearer <token>`, or from a `?token=` parameter for clients that cannot set headers. That fallback exists for the Ling client later in this chapter. It has a real cost: tokens in URLs end up in proxy logs and browser history. Keep the fallback on loopback and TLS-only deployments, or delete it if your clients can send headers.

Validation lives at the edge

`create_note` trims the title, rejects an empty one with `400`, a title over 80 characters with `400`, and a body over 2,000 characters with `413`. Limits matter in a single-threaded server for a reason beyond tidiness: a 30 MiB body (the runtime accepts up to 32 MiB) is parsed on the interpreter thread while everyone waits. Reject early and keep stored values small. Note `clamp(to_number(...), 1, 100)` on `limit`: `to_number("abc")` is `0`, and clamping turns every odd input into something safe.

Sessions are rows

`login` verifies the password with `password_verify` against an Argon2id hash made by `password_hash`, mints a token with `random_hex(16)` (the OS random generator, not the game-logic `rand`), and stores `(token, user, expires)` in SQLite. Because the session is a row, a restart does not log anybody out, and revoking a session is a `DELETE`. Argon2id is deliberately slow: the runtime fixes its parameters at 64 MiB of memory, 3 passes and 4 lanes, so each hash costs real time and memory. That is exactly what you want against password guessing and exactly why the next feature exists.

A login throttle in twenty lines

`login` counts failed attempts for the user name in the last 300 seconds and refuses with `429` after five. The attempt row is written only on failure and the count is checked before any hashing happens, so a flood of bad guesses cannot make the server burn memory on Argon2. Here it is in action:

```
OUTPUT
$ for i in 1 2 3 4 5 6; do curl -s -w ' [%{http_code}]\n' -X POST -d
'user=ada&password=guess' $H/login; done
{"error":"bad credentials"} [401]
{"error":"bad credentials"} [401]
{"error":"bad credentials"} [401]
{"error":"bad credentials"} [401]
{"error":"bad credentials"} [401]
{"error":"too many attempts, try again later"} [429]
$ curl -s -w ' [%{http_code}]\n' -X POST -d 'user=ada&password=lovelace-1815' $H/login
{"error":"too many attempts, try again later"} [429]
```

The last line is the trade-off, not a bug: an attacker who guesses wrongly five times locks the *real* Ada out for five minutes. Keying on the name prevents distributed guessing; keying on `client_ip` would not lock the victim out but, as we saw, that header is forgeable unless a trusted proxy sets it. For a small internal service the name key is the safer default. For a public one, throttle on both, and tell users what happened.

> **TIP**
>
> Seeding users from `cli_arg("ada_pw", ...)` with `INSERT OR IGNORE` is fine for a demo. In a real deployment read secrets from the environment with `env_get("ADA_PW", "")`, refuse to start if it is empty, and never put a password on a command line, where every user on the machine can see it in the process list.

Exercising the service

Always test an API from outside, with a tool that is not your own client. This is the transcript of a session with `curl` (tokens shortened):

```
OUTPUT
$ curl -s -i $H/notes
HTTP/1.1 401 Unauthorized
{"error":"login required"}

$ curl -s -X POST -d "user=ada&password=lovelace-1815" $H/login
{"token":"<token>","expires_in":3600}

$ curl -s -i -X POST -H "Authorization: Bearer $T" -d "title=Buy milk&body=2 litres"
$H/notes
HTTP/1.1 201 Created
{"id":1,"title":"Buy milk","body":"2 litres"}

$ curl -s -i -X POST -H "Authorization: Bearer $T" -d "title=%20%20%20" $H/notes
HTTP/1.1 400 Bad Request
{"error":"title is required"}

$ curl -s -i -X POST -H "Authorization: Bearer $T" --data-urlencode "title=big" --data-
urlencode "body=<2,100 x characters>" $H/notes
HTTP/1.1 413 Payload Too Large
{"error":"body too large (max 2000)"}

$ curl -s -H "Authorization: Bearer $T" "$H/notes?limit=1"
[{"id":1,"title":"Buy milk","body":"2 litres"}]

$ curl -s -i -X DELETE -H "Authorization: Bearer $T" "$H/note?id=1"
HTTP/1.1 200 OK
{"deleted":1}

$ curl -s -i -X PUT $H/notes
HTTP/1.1 404 Not Found
not found
```

Two observations. The server's own log recorded exactly the refusals you triggered, which is the point of `fail`:

```
OUTPUT
warn status=401 login required
info login user=ada
info create user=ada id=1
warn status=400 title is required
warn status=413 body too large (max 2000)
```

And the last request shows a seam: an unrouted method or path gets the runtime's plain-text `404 not found`, not your JSON error and not `405`. Clients that parse every error as JSON must tolerate that. If it matters to you, register a catch-all route behind a reverse proxy that rewrites 404 bodies.

Ownership isolation also held up. Logged in as `bob`, asking for Ada's note with `GET /note?id=1` returned `{"error":"not found"}` with a 404, exactly as it would for an id that never existed.

> **TRY IT**
>
> Start the service and use two different users to confirm that Bob cannot see or delete Ada's notes, then restart with `--db notes.db` and confirm that tokens and notes both survive. Bob's password is `bob-pass-42`.

Persistence and restarts

Pass `--db notes.db` and the same code uses a file. `db_open` turns on write-ahead logging and foreign keys, so you will see `notes.db`, `notes.db-wal` and `notes.db-shm` next to each other; back up all three together, or run `PRAGMA wal_checkpoint` first. The schema statements use `CREATE TABLE IF NOT EXISTS` and the seeds `INSERT OR IGNORE`, which makes start-up idempotent. In a test run we created a note, killed the process, started it again and listed notes with the token from the first run:

```
OUTPUT
{"id":1,"title":"survives restarts","body":""}
[{"id":1,"title":"survives restarts","body":""}]
```

Schema changes need discipline because there is no migration tool in the runtime. Keep a `schema_version` row and apply numbered `db_exec` steps at start-up, or move to the Rust `Db::run_migrations` in `ling-http` once the schema is worth protecting.

One quirk to remember: `db_exec` documents `REAL` binding, but a whole number binds as an `INTEGER`. SQLite's type affinity usually hides this, and our `expires REAL` column compares fine, but compare against the column's declared type in your own schemas rather than the bound value's.

A client in Ling

Ling's HTTP client is small: `http_get(url)` blocks and returns the body (check `http_get_status()` afterwards), and `http_post_async(url, body, content_type)` starts a POST in the background, which you poll with `http_job_poll`. Neither can set a request header, which is why the service accepts `?token=`. The client below logs in, creates a note, lists notes, and then shows the two failure modes a client must handle: a clean `401` and a connection that never happened.

GUI

```
// Tiny client for the notes service. Ling's HTTP client can GET and POST
// but cannot set request headers, so the token travels in ?token=.
fn post_form(url, body) {
  bind job = http_post_async(url, body, "application/x-www-form-urlencoded")
  bind r = ""
  bind n = 0
  while r == "" && n < 200 {
    bind r = http_job_poll(job)
    sleep(25)
    bind n = n + 1
  }
  return r
}

// Pull a string value out of a flat JSON object. Good enough for our own API.
fn json_field(text, name) {
  bind key = "\"" + name + "\":\""
  bind at = str_find(text, key)
  if at < 0 { return "" }
  bind rest = substr(text, at + len(key), len(text))
  return substr(rest, 0, str_find(rest, "\""))
}

bind start = do {
  bind base = cli_arg("base", "http://127.0.0.1:18110")
  bind reply = post_form(base + "/login", "user=ada&password=lovelace-1815")
  println("login reply: ", substr(reply, 0, 20), "...")
  bind token = json_field(reply, "token")
  println("token length: ", len(token))

  println(post_form(base + "/notes?token=" + token, "title=From the client&body=sent
via http_post_async"))
  println(http_get(base + "/notes?token=" + token))
  println("status: ", http_get_status())
  println(http_get(base + "/notes?token=nope"))
  println("status: ", http_get_status())
```

```
  bind dead = http_get("http://127.0.0.1:1/")
  println("unreachable status: ", http_get_status(), " body empty: ", dead == "")
}
```

```
OUTPUT
login reply: {"token":"1211dd2ee4...
token length: 32
{"id":1,"title":"From the client","body":"sent via http_post_async"}
[{"id":1,"title":"From the client","body":"sent via http_post_async"}]
status: 200
{"error":"login required"}
status: 401
unreachable status: 0 body empty: true
```

Three details are worth copying into your own clients.

Status zero is not a status. `http_get_status()` is `0` when nothing answered at all, which is different from every real 4xx or 5xx. Treat `0` as a transport failure and retry or alert; treat 401 as an authorisation problem and do not retry.

Polling needs a bound. `post_form` loops at most 200 times with a 25 ms sleep. An unbounded `while r == ""` would hang forever if the server vanished, because an empty reply and an unfinished job both look like `""`. The same job id returns a JSON `{"error": ...}` string if the request itself failed, so check for it.

Hand-rolled JSON extraction is a trap. `json_field` finds `"token":"` and reads to the next quote. That is acceptable for a flat object whose shape you control. It would break on escaped quotes inside a value, and it is the reason to keep your API responses flat. For a client that must handle arbitrary JSON, write it in Rust with `serde_json`, or shell out to a tool.

Hardening: where the runtime stops and you begin

Here is the checklist this chapter was building toward. Each item is backed by something we observed rather than something we hoped.

Catch errors at the handler boundary. `try(function, args...)` works with a *named* function and returns `ok(v)` or `bad(e)`. Wrap risky handlers so the client never sees internals:

```
GUI
form Response { status, content_type, body }

fn risky(req) {
  db_exec("INSERT INTO missing_table VALUES (1)")
  return "unreachable"
}

fn guarded(req) {
  bind r = try(risky, req)
  return match r {
    ok(v) => v,
    bad(e) => Response(500, "application/json", "{\"error\":\"internal error\"}")
  }
}

bind start = do {
  db_open(":memory:")
  http_route("GET", "/risky", guarded)
  http_serve("127.0.0.1", 18113)
}
```

```
OUTPUT
$ curl -s -i http://127.0.0.1:18113/risky
HTTP/1.1 500 Internal Server Error
{"error":"internal error"}
```

Pass a closure literal to `try` and you get `()` instead of a result, so keep the guarded function named. Register `guarded` for each route and keep the real work in `risky`-style functions.

Headers. The runtime adds only `X-Frame-Options: DENY` and a `Content-Security-Policy` with `frame-ancestors 'none'`. It sends no `Strict-Transport-Security`, no `X-Content-Type-Options`, no `Cache-Control`. Add them at the reverse proxy. The response struct lets you set status, body, content type, a `set_cookie` string and a `location`, but not arbitrary headers.

Transport. Use `http_serve_tls(host, port, cert_path, key_path)`, whose last two arguments are file *paths* despite the runtime's internal names, or run plain `http_serve` on `127.0.0.1` behind a TLS-terminating proxy. `http_serve_dev_tls` generates a throwaway certificate; it is for `curl -k` on your laptop.

Identity of the caller. Do not trust `client_ip` unless a proxy you control is the only way in and overwrites `X-Forwarded-For`. Key throttles on accounts or API keys first.

Concurrency. Keep handlers under a few milliseconds. Do outbound calls with `http_post_async` and poll from a later request rather than blocking inside one. If you need parallel throughput, run several processes on different ports behind a proxy, each with its own database file, or move the hot path to Rust.

CORS. The Ling runtime sets no CORS headers (we sent `Origin: http://evil.example` and got none back, and a preflight `OPTIONS` to `/notes` returned the plain `404`), so a browser on another origin cannot call your API. That is the safe default; add the headers deliberately at the proxy for the origins you trust. (The Rust `ling_http::App` does the opposite: it installs permissive CORS. See the next chapter.)

Secrets and keys. Passwords from `env_get`, never argv. Keep the SQLite file, and the `.lingtp` and `.ling-ssh` host keys if you use them, out of web roots and backed up.

Logging. There is no access log. The `fail` helper above is a start; add an `info` line per mutating request with the user and the id. Do not log tokens, passwords or request bodies.

Weaknesses we chose to leave. The login path hashes only when the user exists, so response time reveals whether a name is registered. The five-attempt lock can be abused to deny a user access. Neither matters for an internal tool; both matter on the public internet. Knowing which of your weaknesses you accepted is part of the job.

Deployment

A minimal, boring deployment is one process per service, supervised by the init system, listening on loopback, with a proxy in front. The unit below is a sketch: it is not something we could run in this environment, so treat it as a starting point you must test.

```
OUTPUT
[Service]
ExecStart=/usr/local/bin/ling run /srv/notes/svc.ling --port 18110 --db
/var/lib/notes/notes.db
Restart=on-failure
User=notes
```

Before you call it done, run this checklist against the live process: refuse without a token; refuse an expired token; confirm a deleted note stays deleted after a restart; confirm the process answers `/health` after `kill -9` and a supervised restart; read the log and check

there are no secrets in it. Then do it again through the proxy, because that is the path your users take.

Chapter summary

- The Ling HTTP server accepts connections concurrently but runs handlers one at a time on the interpreter thread: simple and race-free, but a slow handler delays every request.
- Routes match method plus exact path. Pass identifiers in the query string, parse form bodies with `query_param`, build JSON by hand with `json_escape`, and always use SQL parameters.
- Return a `Response` form for any status other than 200. Centralise error bodies in one `fail` helper so the shape and the log line never drift.
- Authenticate with `password_hash` / `password_verify` and server-side token rows; mint tokens with `random_hex`; throttle failed logins before hashing.
- An uncaught error becomes a `500` that leaks internals, and `client_ip` is spoofable. Wrap handlers with `try` on a named function and place the service behind a proxy you control.
- The Ling client can `GET` and `POST` but cannot set headers; treat `http_get_status() == 0` as transport failure and always bound your polling loops.
- Add what the runtime omits (security headers, HSTS, access logs, CORS policy) at the reverse proxy.

Exercises

1. **Add `PUT /note?id=N`.** Let a user change a note's title and body. Validate with the same limits as creation, scope the `UPDATE` to the owner, and return `404` when no row changed. Which status should an empty new title produce?
2. **Expire sessions.** The `sessions` table grows forever. Add a `POST /logout` route that deletes the caller's token, and make `login` delete all expired rows first. Verify with `curl` that a token stops working after logout.
3. **Offload a slow job.** Add a route `GET /slow-report` that must call another local service at `http://127.0.0.1:18200/report` (assume it takes a second). Without blocking the server, how would you design it with `http_post_async` and `http_job_poll`? Sketch the two routes and say what the client sees.

Answers

1. Add `http_route("PUT", "/note", update_note)`. The handler runs the same `current_user` guard, reads `title` and `body` from `req.body`, validates exactly as `create_note` does, then runs `db_exec("UPDATE notes SET title = ?1, body = ?2 WHERE id = ?3 AND owner = ?4", title, body, id, who)`. `db_exec` returns the number of rows changed, so `if n == 0 { return fail(404, "not found") }` covers both a missing id and someone else's note. Return the updated row with `db_query` and `note_json`. An empty or whitespace-only title is a client error in the request itself, so `400`, as on creation. A `404` would imply the note is missing, which is a different problem.
2. `logout` calls `current_user` (or reads the token the same way) and then `db_exec("DELETE FROM sessions WHERE token = ?1", token)`, returning `{"ok":true}`. At the start of `login`, run `db_exec("DELETE FROM sessions WHERE expires < ?1", epoch_now())`. Because `current_user` already filters on `expires > now`, expired tokens are refused even before cleanup; cleanup only bounds table size. After logout, a `curl` with the old token must return `401 login required`.
3. Use a two-step design. `POST /reports` starts the job: `bind job = http_post_async("http://127.0.0.1:18200/report", "", "application/json")`, store `job` in a `jobs` table with the owner and a `pending` state, and return `202` with the job id immediately. `GET /reports?id=...` looks the job up and calls `http_job_poll(job)`; while it returns `""` answer `{"state":"running"}`, and when it returns text, store it and answer with the result. The server thread is never blocked for more than the time of a poll. The client sees one fast `202` and then polls, which is the same shape as a hosted job API. Remember that `""` can also mean "finished with an empty body", so record an explicit state rather than inferring it from the poll alone.

CHAPTER 4

Web Apps with the Rust Crates

What you'll learn

The previous chapter built a JSON service in Ling itself. This one steps down a level to the Rust crates that Ling's own HTTP runtime is built on, `ling-http` and `ling-web`, and uses them directly to build something Ling's built-ins cannot do: a server-rendered web application with accounts, server-side sessions, CSRF protection, flash messages, inherited templates, URL-grouped blueprints and a JSON API, all in about 300 lines of Rust.

You will:

- decide, with evidence, when a project belongs in Ling and when it belongs in Rust;
- set up a Cargo project that depends on `ling-http`, `ling-web` and `ling-crypto`;
- build and run a complete application, `linkshelf`, and exercise it end to end with `curl`;
- understand each piece: the `Db` pool, the session layer, CSRF tokens, templates, blueprints and your own middleware;
- avoid three traps we hit while building it, including one in `ling_http::App` that silently drops its middleware.

This chapter assumes you can read Rust: `async fn`, `?`, traits and derive macros. You do not need to know `axum` in advance; every extractor we use is explained where it appears. Everything below was compiled and run: Rust 1.98 with `cargo build` in a clean project, then driven with `curl` on loopback.

Ling or Rust: choosing honestly

Both stacks run the same underlying code. `http_serve` in Ling calls `ling_http::serve_http`, `db_open` opens a `rusqlite` connection through the same re-export, and the Ling password builtins call `ling-crypto`. So the choice is not about speed of the hello-world path. We measured 1,000 sequential keep-alive requests to a trivial health route: the Ling service took about 0.45 seconds and the Rust service, built in *debug* mode, about 0.6 seconds. Handler dispatch is not your bottleneck in either.

The real differences are structural:

Concern	Ling services	Rust with `ling-http` / `ling-web`
Concurrency	One handler at a time on the interpreter thread	Async handlers across a thread pool; slow work awaits without blocking others
JSON in	No parser; form-encoded bodies only	`serde` structs, validated at the type level
JSON out	Hand-built strings plus `json_escape`	`serde_json`, `Json<T>`
Sessions, CSRF, flash, templates	Build them yourself from `db_*` and `cookie_get`	`ling-web` provides them
Response headers	Only status, type, cookie, location	Any header, any middleware
Static typing	None; typos are silent `()`	The compiler checks it
Iteration speed	Edit and `ling run`	Compile cycles

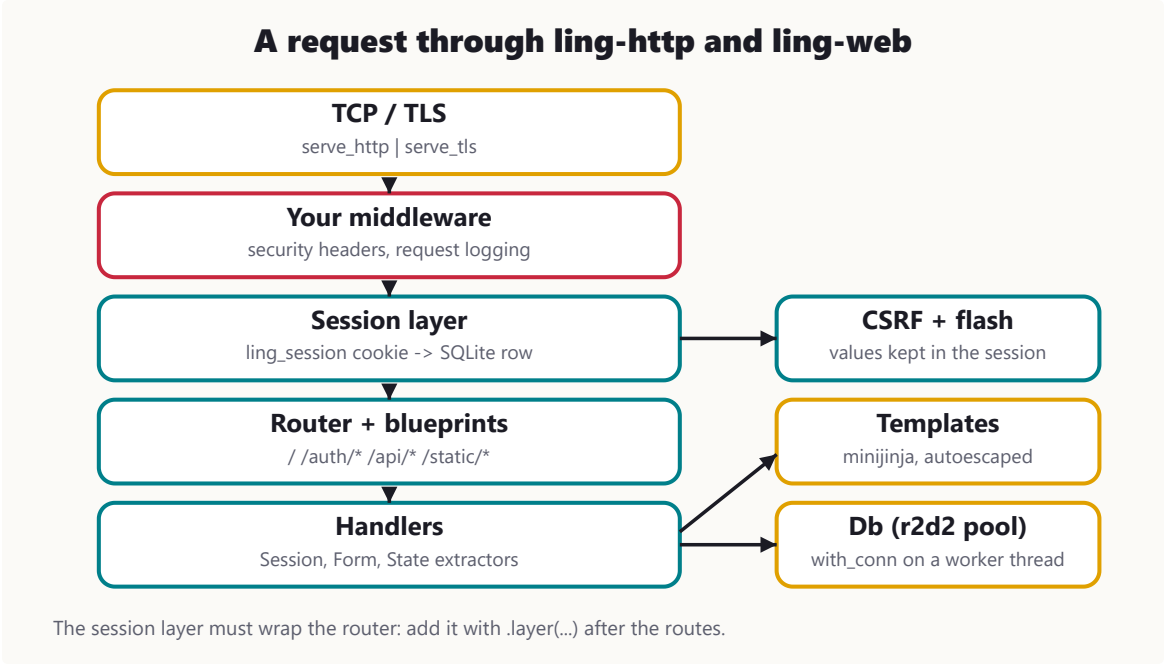
The rule of thumb we use: if the service is a thin API in front of SQLite, has a handful of routes and one operator, write it in Ling. Reach for Rust when you need a browser-facing application with sessions and templates, JSON bodies of real shape, outbound calls from handlers, more than one request in flight, or a type system standing between you and a 3 a.m. page. Moving later is cheap in one direction: your SQL, your route table and your design notes carry over. If you can already see that you will cross a line in the table, start in Rust.

> NOTE
>
> Ling programs cannot call into `ling-web`; it has no built-ins. The reverse is also true: a Rust service cannot run a `.ling` file as a handler. They share crates and conventions, not a runtime. The embedding crate `ling-lang` exists for running Ling from Rust; see *Using the Crates from Rust*.

What the crates give you

`ling-http` is the base layer: an `axum` router toolkit with a pooled SQLite handle (`Db`), a uniform error type (`HttpError`), a `Resource` trait for CRUD routes, and three ways to serve (`serve_http`, `serve_tls`, `serve_dev_tls` behind the `dev-certs` feature). It re-exports `axum`, `rusqlite`, `serde_json` and `tokio`, so you can use `ling_http::axum::...` and never hit a version mismatch between your `axum` and the crate's.

`ling-web` sits on top and adds what a browser-facing application needs.



The layers a request crosses in the application of this chapter. Layers added later with `.layer(...)` sit outside earlier ones.

Item	What it does
`session_layer(&db, secure)`	SQLite-backed sessions: an opaque `ling_session` cookie, `HttpOnly`, `SameSite=Lax`, seven-day sliding expiry. The data is server-side, so deleting the row revokes the session.
`Session`	The extractor you take in a handler: `get`, `insert`, `remove`, `cycle_id`, `flush`.
`csrf::csrf_token`, `csrf::verify_csrf`	A 256-bit per-session token and a constant-time check.
`flash::flash`, `flash::take_flashes`	One-shot messages for the next page.
`Templates::load(dir)`	`minijinja` templates with inheritance, loops and automatic HTML escaping.
`blueprint(prefix, router)`	A named `Router::nest`, for grouping routes.
`serve_static(prefix, dir)`	A static-file mount.

Before writing anything, run what ships. Both repository examples compile in the workspace; this is the CRUD example against a live server, exactly as built (`hello`, port 18140):

```
OUTPUT
$ curl -s $H/health
{"service":"ling-http/hello","status":"ok"}
$ curl -s -X POST $H/notes -H 'content-type: application/json' -d
'{"title":"hi","body":"first note"}'
{"id":1,"title":"hi","body":"first note","created_at":"2026-10-29 00:26:24"}
$ curl -s -X PUT $H/notes/1 -H 'content-type: application/json' -d '{"title":"edited"}'
{"id":1,"title":"edited","body":"first note","created_at":"2026-10-29 00:26:24"}
$ curl -s $H/notes/99
{"error":"database error: Query returned no rows"}
$ curl -s -X POST $H/notes -H 'content-type: application/json' -d '{"body":"no title"}'
Failed to deserialize the JSON body into the target type: missing field `title` at line
1 column 19
```

Look at the last two lines closely. The `404` body is a *database error message*, which is an information leak and also an ugly API. And a body that fails to deserialise is rejected by `axum` before your code runs, in plain text, not in the `{"error": ...}` shape. `HttpError` is a fine starting point; for a public API you will want your own error type, and we build one below.

Setting up the project

Create a binary crate and depend on the crates by path (adjust the paths to your checkout of the repository). The empty `[workspace]` table keeps Cargo from treating your project as part of another workspace.

```
[package]
name = "linkshelf"
version = "0.1.0"
edition = "2021"

[dependencies]
ling-http   = { path = "C:/Users/User/Programs/ling/crates/ling-http" }
ling-web    = { path = "C:/Users/User/Programs/ling/crates/ling-web" }
ling-crypto = { path = "C:/Users/User/Programs/ling/crates/ling-crypto" }
tokio       = { version = "1", features = ["rt-multi-thread", "macros"] }
serde       = { version = "1", features = ["derive"] }
minijinja   = "2"
anyhow      = "1"

[workspace]
```

That built cleanly in about two and a half minutes the first time and in seconds afterwards. `ling-crypto` is there for one reason: Argon2id password hashing, the same implementation behind Ling's `password_hash` builtin. Passwords hashed in a Ling service and in this Rust service are interchangeable PHC strings.

The application has three templates in `templates/`: a base layout, a home page and a shared login/register page. This is the base, to show template inheritance and escaping in one go:

```
OUTPUT
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>{% block title %}Linkshelf{% endblock %}</title>
  <link rel="stylesheet" href="/static/app.css">
</head>
<body>
  <header>
    <a href="/">Linkshelf</a>
    {% if user %}
      <form method="post" action="/auth/logout" class="inline">
        <input type="hidden" name="csrf_token" value="{{ csrf_token }}">
        <span>{{ user }}</span> <button>Log out</button>
      </form>
    {% endif %}
  </header>
  {% for f in flashes %}<p class="flash {{ f.category }}">{{ f.message }}</p>{% endfor
%}
  {% block content %}{% endblock %}
</body>
</html>
```

Logging out is a `POST` with a CSRF token, not a link, because a `GET` that changes state can be triggered by any page that embeds an image tag pointing at it.

The application, part by part

The listings below are excerpts of the single `src/main.rs` that was compiled and run (309 lines); the order is the order in the file, and nothing in the excerpts was edited.

State, errors and the cost of leaking

```
const MIGRATIONS: &[(&str, &str)] = &[(
    "0001_init",
    "CREATE TABLE users(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL
UNIQUE, hash TEXT NOT NULL);
     CREATE TABLE links(id INTEGER PRIMARY KEY AUTOINCREMENT,
                        user_id INTEGER NOT NULL REFERENCES users(id),
                        url TEXT NOT NULL, title TEXT NOT NULL,
                        created_at TEXT NOT NULL DEFAULT (datetime('now')));",
)];

#[derive(Clone)]
struct AppState {
    db: Db,
    templates: Templates,
}

/// What the browser (or API client) is allowed to learn about a failure.
/// Details go to stderr; the response body stays generic.
enum AppError {
    Unauthorized,
    NotFound,
    Internal,
}
```

`Db` is a pooled, clonable handle, and the application state is just `Db` plus a `Templates` handle (an `Arc` inside), so cloning state per request is cheap. Migrations are `(name, sql)` pairs; `run_migrations` records applied names in a `_migrations` table and skips them next time.

Next comes the error type, which is our answer to the leak we saw in the shipped example:

```
impl From<HttpError> for AppError {
    fn from(e: HttpError) -> Self {
        match e {
            HttpError::NotFound => AppError::NotFound,
            other => {
                eprintln!("internal error: {other}");
                AppError::Internal
            }
        }
    }
}

impl From<anyhow::Error> for AppError {
    fn from(e: anyhow::Error) -> Self {
        eprintln!("internal error: {e}");
        AppError::Internal
    }
}

impl IntoResponse for AppError {
    fn into_response(self) -> Response {
        match self {
            AppError::Unauthorized => (StatusCode::UNAUTHORIZED, "login
required").into_response(),
            AppError::NotFound => (StatusCode::NOT_FOUND, "not found").into_response(),
            AppError::Internal => (StatusCode::INTERNAL_SERVER_ERROR, "something went
wrong").into_response(),
        }
    }
}

type Res<T> = Result<T, AppError>;
```

Every handler returns `Result<T, AppError>`. The `From` implementations mean that `?` works on `HttpError` and on `anyhow::Error`, and both collapse to a generic `500 something went wrong` after logging the real error to standard error. A genuinely missing row becomes a `404`. Nothing from `rusqlite` ever reaches a browser.

Your own middleware

`ling_http::App::new()` is documented as an `axum` router with tracing, a 60 second timeout, a 25 MiB body limit and permissive CORS. We do not use it, for a reason found by experiment. In axum, `Router::layer` wraps only the routes that exist *at the moment you call it*. `App::new()` applies its layers to an empty router, and your routes arrive afterwards through `.merge(...)`, so they are not wrapped. We tested the CORS layer on the shipped example: a request carrying `Origin: http://evil.example` got no `access-control-*` headers back, and a preflight `OPTIONS` got `405 Method Not Allowed`. The timeout, body limit and tracing layers follow the same rule, so do not assume they protect you; we only verified the CORS case directly.

> WARNING
>
> If you build on `App::new()`, add your own `TimeoutLayer`, body limit and CORS policy on the *final* router, after all routes have been merged. Do not trust the constructor's defaults to apply to your handlers.

Writing our own is short. This one adds three security headers to every response:

```
async fn security_headers(req: Request, next: Next) -> Response {
    let mut resp = next.run(req).await;
    let h = resp.headers_mut();
    h.insert(header::X_CONTENT_TYPE_OPTIONS, HeaderValue::from_static("nosniff"));
    h.insert(header::X_FRAME_OPTIONS, HeaderValue::from_static("DENY"));
    h.insert(
        header::CONTENT_SECURITY_POLICY,
        HeaderValue::from_static("default-src 'self'; frame-ancestors 'none'"),
    );
    resp
}
```

It is installed with `.layer(middleware::from_fn(security_headers))` as the *last* layer, which makes it the outermost, so it also decorates the responses that the session layer and static-file service produce, and even the `405` for a wrong method. Compare this with Ling's runtime, which can add only a fixed pair of headers.

Sessions, CSRF and flashes in three helpers

```
async fn session_user(session: &Session) -> Option<(i64, String)> {
    let uid = session.get::<i64>("uid").await.ok()??;
    let name = session.get::<String>("name").await.ok()??;
    Some((uid, name))
}

/// Shared page context: CSRF token, pending flash messages, current user.
async fn page_ctx(session: &Session) -> Res<(String, Vec<flash::Flash>,
Option<String>)> {
    let token = csrf::csrf_token(session).await?;
    let flashes = flash::take_flashes(session).await?;
    let user = session_user(session).await.map(|(_, n)| n);
    Ok((token, flashes, user))
}

async fn check_csrf(session: &Session, token: &str) -> Res<bool> {
    Ok(csrf::verify_csrf(session, token).await?)
}

// ---------- pages ----------
```

A `Session` is an extractor, so any handler that lists it as a parameter gets the current session loaded by the layer. We store the user's id and name under two keys when login succeeds; `session_user` reads them back and returns `None` when absent. `page_ctx` gathers what every page template needs: the CSRF token (created on first call and kept in the session), any pending flash messages (taken, so they show once), and the user name.

The CSRF rule is simple and strict: every state-changing form carries the hidden token, and every `POST` handler verifies it before doing anything else. `verify_csrf` returns `false`, not an error, for a missing or wrong token, so you decide what rejection looks like. Here is a form handler that does it, with validation and a flash on failure:

```
async fn add_link(State(st): State<AppState>, session: Session, Form(f):
Form<LinkForm>) -> Res<Redirect> {
    let Some((uid, _)) = session_user(&session).await else { return
Err(AppError::Unauthorized) };
    if !check_csrf(&session, &f.csrf_token).await? {
        flash::flash(&session, "error", "That form expired - try again.").await?;
        return Ok(Redirect::to("/"));
    }
    let (title, url) = (f.title.trim().to_string(), f.url.trim().to_string());
    let valid_url = url.starts_with("https://") || url.starts_with("http://");
    if title.is_empty() || title.len() > 80 || !valid_url || url.len() > 500 {
        flash::flash(&session, "error", "Need a title (80 chars max) and an http(s)
URL.").await?;
        return Ok(Redirect::to("/"));
    }
    st.db
        .with_conn(move |c| {
            c.execute(
                "INSERT INTO links(user_id, title, url) VALUES (?1, ?2, ?3)",
                ling_http::rusqlite::params![uid, title, url],
            )
        })
        .await?;
    flash::flash(&session, "success", "Saved.").await?;
    Ok(Redirect::to("/"))
}
```

Notice the order: session check, CSRF check, validation, write, flash, redirect. The handler never renders on `POST`; it redirects to a `GET`. This is the post/redirect/get pattern, and it means a browser refresh cannot resubmit the form. Notice also that the URL is checked with `starts_with("https://")` or `starts_with("http://")`. A `javascript:` URL in an `href` is executable even when HTML escaping is perfect, so the escaping in the template protects you from markup injection and this check protects you from scheme injection. Both are needed.

Login, and the one line that defeats session fixation

```
async fn login(State(st): State<AppState>, session: Session, Form(f): Form<AuthForm>) -
> Res<Redirect> {
    if !check_csrf(&session, &f.csrf_token).await? {
        return Err(AppError::Unauthorized);
    }
    let name = f.name.trim().to_string();
    let lookup = name.clone();
    let row = st
        .db
        .with_conn(move |c| {
            c.query_row("SELECT id, hash FROM users WHERE name = ?1", [lookup], |r| {
                Ok((r.get::<_, i64>(0)?, r.get::<_, String>(1)?))
            })
        })
        .await;
    let ok_uid = match row {
        Ok((uid, hash)) if Argon2idParams::verify_password(f.password.as_bytes(),
&hash).is_ok() => Some(uid),
        _ => None,
    };
    let Some(uid) = ok_uid else {
        flash::flash(&session, "error", "Wrong name or password.").await?;
        return Ok(Redirect::to("/auth/login"));
    };
    // Fresh session id after a privilege change defeats session fixation.
    session.cycle_id().await.map_err(|e| anyhow::anyhow!(e))?;
    session.insert("uid", uid).await.map_err(|e| anyhow::anyhow!(e))?;
    session.insert("name", name).await.map_err(|e| anyhow::anyhow!(e))?;
    Ok(Redirect::to("/"))
}
```

The password check runs Argon2id's `verify_password`. If it fails, the user sees the same flash for a wrong name or a wrong password. After success, `session.cycle_id()` replaces the session identifier before we store the user: an attacker who planted a known session id in the victim's browser before login gets an id that is now dead. We observed it work: the cookie value before login and after login differ.

Blueprints and `main`

```
#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let port: u16 = std::env::var("PORT").ok().and_then(|p|
p.parse().ok()).unwrap_or(18120);
    let secure = std::env::var("LINKSHELF_SECURE").is_ok();
    let db = match std::env::var("LINKSHELF_DB") {
        Ok(path) => Db::connect(path).await?,
        Err(_) => Db::connect_memory().await?,
    };
    db.run_migrations(MIGRATIONS).await?;
    let sessions = ling_web::session_layer(&db, secure).await?;
    let state = AppState { db, templates: Templates::load("templates")? };

    let auth = Router::new()
        .route("/login", get(login_form).post(login))
        .route("/register", get(register_form).post(register))
        .route("/logout", post(logout));
    let api = Router::new().route("/links", get(api_links));

    let app = Router::new()
        .route("/", get(home))
        .route("/links", post(add_link))
        .route("/links/{id}/delete", post(delete_link))
        .route("/healthz", get(|| async { "ok" }))
        .merge(ling_web::blueprint("/auth", auth))
        .merge(ling_web::blueprint("/api", api))
        .merge(ling_web::serve_static("/static", "static"))
        .with_state(state)
        .layer(sessions)
        .layer(middleware::from_fn(security_headers));

    ling_http::serve_http(app, format!("127.0.0.1:{port}").parse()?).await?;
    Ok(())
}
```

The `auth` and `api` routers are built as separate units and mounted with `ling_web::blueprint("/auth", auth)`, which is `Router::nest` under a name. This is how a large application stays legible: each blueprint can live in its own module with its own tests, and the top level is a table of contents. Route parameters use brace syntax in axum 0.8 (`/links/{id}/delete`) and arrive through the `Path` extractor, already parsed into an `i64`: no hand-written `to_number`, and a non-numeric id never reaches your handler.

Look at the layer order at the bottom, because it is the most common mistake in these applications:

```
FRAGMENT
routes ... .with_state(state)
        .layer(sessions)                          // wraps the routes above
        .layer(middleware::from_fn(security_headers))  // wraps the session layer too
```

The session layer must be added *after* the routes it serves, or handlers that take `Session` fail at request time. Your own middleware should be added after that, so that it sees every response.

Running it, and attacking it

Build and run with `cargo run` from the project directory (templates and static files are read from relative paths, so the working directory matters). Configuration is by environment variables: `PORT`, `LINKSHELF_DB` for a database file (default is a private in-memory database) and `LINKSHELF_SECURE` to mark cookies `Secure`. This is the full session captured with a cookie jar, first as a stranger, then as a user who tries to misbehave:

```
OUTPUT
$ GET / (no session)
HTTP/1.1 303 See Other location: /auth/login
$ POST /auth/register, password too short
HTTP/1.1 303 See Other location: /auth/register
$ POST /auth/register, forged CSRF token
HTTP/1.1 401 Unauthorized
$ POST /auth/register, valid
HTTP/1.1 303 See Other location: /auth/login
$ POST /auth/login, wrong password
HTTP/1.1 303 See Other location: /auth/login
<p class="flash error">Wrong name or password.</p>
$ POST /auth/login, right password
HTTP/1.1 303 See Other location: /
session cookie before login: MWngWU...  after login: LxvgaK...
$ POST /links (title + http URL)
HTTP/1.1 303 See Other location: /
$ POST /links (javascript: URL)
HTTP/1.1 303 See Other location: /
<p class="flash success">Saved.</p>
<p class="flash error">Need a title (80 chars max) and an http(s) URL.</p>
<li><a href="https:&#x2f;&#x2f;example.org&#x2f;&lt;b&gt;" rel="noopener
noreferrer">Ling book</a>
$ GET /api/links (session cookie)
[{"id":1,"title":"Ling book","url":"https://example.org/<b>"}]
$ GET /api/links (no cookie)
HTTP/1.1 401 Unauthorized login required
$ POST /links/1/delete, bad token
HTTP/1.1 401 Unauthorized
$ POST /links/1/delete, good token
HTTP/1.1 303 See Other location: /
$ POST /links/1/delete again
HTTP/1.1 404 Not Found
$ POST /auth/logout
HTTP/1.1 303 See Other location: /auth/login
$ GET /api/links after logout
HTTP/1.1 401 Unauthorized
$ GET /healthz, headers
```

```
HTTP/1.1 200 OK
x-content-type-options: nosniff
x-frame-options: DENY
content-security-policy: default-src 'self'; frame-ancestors 'none'
```

Read the transcript as a checklist:

- A stranger on `/` is redirected, not served.
- A short password is refused with a flash; a forged CSRF token gets a `401` and no side effect.
- The flash from a failed login appeared on the *next* page and was not repeated: flash messages are one-shot.
- The session cookie value changed at login (`MWngWU...` to `LxvgaK...` in this run).
- The `javascript:` URL was refused, and the valid link was rendered with the `/` and `<` characters of its URL escaped by the template engine, so `<b>` could not become markup.
- The JSON API answered the same session cookie, and answered `401` without it.
- Deleting with a bad token did nothing; deleting twice yields `404` the second time, because the `DELETE` is scoped by both id and owner.
- After logout the cookie is dead: `flush` removed the session row on the server.
- Every response, including this plain-text one, carried our three headers.

> TIP
>
> Start the app with `LINKSHELF_SECURE=1` and inspect `Set-Cookie`: it gains the `Secure` attribute (we saw `ling_session=...; HttpOnly; SameSite=Lax; Secure; Path=/; Max-Age=604800`). Over plain HTTP a browser will then refuse to keep the cookie, so you will "never log in" in local development. That is not a bug: use `Secure` only when serving HTTPS (`serve_tls` or a TLS proxy), and plain cookies on `localhost`.

What to harden next

This application is a sound skeleton, not a finished product. Here is the honest list:

1. **User enumeration by timing.** For an unknown name the handler skips Argon2 and answers fast, so response time reveals which names exist. Fix by verifying against a dummy hash for unknown names.
2. **No rate limiting.** Add a limiter keyed on the account and the client address (from a trusted proxy), as in the previous chapter's throttle but in a `tower` layer.

3. **No timeout or body-limit layer.** Add `TimeoutLayer` and `RequestBodyLimitLayer` from `tower-http` on the final router (see the warning above).
4. **CSRF token is not rotated at login.** The session id is cycled, but the token stored in the session stays. Rotate it: remove the `_csrf` key at login.
5. **Cookie security.** Run behind TLS with `LINKSHELF_SECURE=1`. Consider `SameSite=Strict` for an app with no cross-site entry links; the layer fixes `Lax`, so customising means building the `SessionManagerLayer` yourself.
6. **Content-Security-Policy.** We set `default-src 'self'`, which blocks inline scripts. If you add scripts, serve them as files from `/static`.
7. **Account lifecycle.** There is no password change, no lockout and no session listing. Because sessions are rows in SQLite, "log out everywhere" is a `DELETE` on the sessions table.

> WARNING
>
> The `ling-web` session store is hand-written over `ling-http`'s `Db`. It is small and readable (about 150 lines in `session_store.rs`), which is a virtue for an audit. Read it before you rely on it: an expired session row is deleted only when its cookie shows up again, so sessions that are simply abandoned accumulate. If your sessions table matters, add a periodic `DELETE FROM sessions WHERE expiry_date < strftime('%s','now')`.

Deployment shape

The pattern is the same as for Ling services, with better tools: one binary, supervised, on loopback, behind a TLS proxy, with the database on a persistent volume. `ling_http::serve_tls(router, addr, cert_path, key_path)` terminates TLS in-process with PEM files if you cannot have a proxy; `serve_dev_tls` needs the `dev-certs` feature and is only for local work. Release builds matter: our timings above were for a debug build, and `cargo build --release` removes the debug-only overhead. Build with `--release`, run as an unprivileged user, give the process a working directory that contains `templates/` and `static/`, and pass configuration through the environment.

Chapter summary

- `ling-http` and `ling-web` are the crates under Ling's own HTTP runtime; use them directly when you need concurrency, typed JSON, sessions, CSRF, templates or custom headers.

- Choose Ling for thin SQLite-backed APIs with few routes; choose Rust for browser-facing applications and anything with more than one request in flight. Trivial-request latency was comparable in our test, so decide on structure, not on speed.
- `Db::with_conn` runs blocking `rusqlite` work on a worker thread; `run_migrations` applies named SQL once.
- `ling-web` supplies SQLite-backed sessions, CSRF tokens, flash messages, `minijinja` templates, blueprints and static files. Add the session layer after your routes and your own middleware after that.
- Define your own error type that logs details and returns generic bodies; `HttpError` exposes database messages and `axum`'s extractor errors are plain text.
- `ling_http::App::new()` applies its layers to an empty router, so merged routes did not get its CORS layer in our test. Apply timeouts, limits and CORS to the final router yourself.
- Cycle the session id at login, verify CSRF before every state change, redirect after `POST`, and validate URL schemes as well as escaping HTML.

Exercises

1. **Rotate the CSRF token at login.** After `session.cycle_id()` in `login`, remove the stored `_csrf` value so that the next page issues a fresh token. Which function from `ling_web::csrf` creates a token on demand, and why does removing the key suffice?
2. **Close the timing leak.** Make `login` take roughly the same time whether or not the user exists. Describe the change and say what you must be careful to hash.
3. **Add a body-limit and timeout layer.** Using `tower-http` (already in `ling-web`'s dependency tree), restrict request bodies to 64 KiB and time out after 10 seconds. Where in `main` does the layer go, and why there?

Answers

1. `ling_web::csrf::csrf_token(&session)` generates and stores a token when the session has none, so deleting the `_csrf` key is enough: the next call to `page_ctx` creates a new one. Call `session.remove::<String>("_csrf").await` after the `cycle_id` call. The key name is an implementation detail of `ling-web`, so a fragile alternative is to build the new session from scratch with `session.flush()` followed by inserting `uid` and `name`; `flush` clears everything including the token. Either way verify by fetching the login page before and after and comparing the hidden field.
2. Keep a precomputed `dummy_hash` (the Argon2id hash of a random string, created at start-up) in the application state. In `login`, when the lookup returns no row, run `Argon2idParams::verify_password(f.password.as_bytes(), &dummy_hash)` and ignore the

result, then show the same flash. Both branches now pay one Argon2id verification. Be careful that the dummy hash uses the same parameters as real ones (use `Argon2idParams::default()` for both), or the timings will differ for a different reason.

3. Add `.layer(RequestBodyLimitLayer::new(64 * 1024))` and `.layer(TimeoutLayer::new(Duration::from_secs(10)))` (or `with_status_code` as `ling-http` does) to the router *after* `.with_state(state)` and the session layer, alongside `security_headers`. They must wrap everything the router serves, and as we saw, a layer applied before a route is merged does not cover that route. Test it by sending a 100 KiB body with `curl --data-binary` and expecting `413`.

CHAPTER 5

Cryptography in Practice

What you'll learn

Ling gives you cryptography as ordinary function calls: `crypto_seal`, `ed25519_sign`, `password_hash`, `totp_verify`. That convenience is the danger. A call that returns plausible-looking hex is not evidence that your system is secure. This chapter teaches you to choose the right primitive for a stated threat, to use it without stepping on the sharp edges this runtime really has, and to assemble the pieces into something you could defend in a design review.

By the end you will have hashed, sealed, signed, verified and key-exchanged with the builtins; reproduced the same results from Rust with the `ling-crypto` crate; and built an encrypted notes vault that detects tampering and deletion. You will also know the list of things the vault does not protect against, which matters more than the list of things it does.

This chapter assumes you have the reference chapter on cryptography at hand for the full builtin list. Here we concentrate on judgement. Transport security appears in *Network Services*; peer-to-peer identity in *Peer-to-Peer and Protocols*.

Begin with the threat, not the algorithm

Before you type a builtin name, write down three things: what you are protecting, from whom, and where the attacker can stand. The table maps the questions you will actually face to the builtin that answers them.

You need to…	Use	Because
Fingerprint data, detect accidental change	`hash_hex` (BLAKE3), `sha256_hex`	Fast, collision resistant, no key
Keep a message secret and tamper-evident	`crypto_seal` / `crypto_open`	XChaCha20-Poly1305, fresh nonce per call
Prove who published something	`ed25519_sign` plus `mldsa87_sign`	Classical and post-quantum; accept only if both verify
Store a login password	`password_hash` / `password_verify`	Argon2id with a random salt
Turn a passphrase into a key	`lw_password_key`	Argon2id with parameters you choose

You need to...	Use	Because
Add a second factor	`totp_*`	Matches every authenticator app
Agree a key with someone	`knot_*` or `lw_kem_*`	X25519 combined with ML-KEM

Notice what is absent. There is no row for "encrypt with a password I typed directly", because a passphrase is not a key, and none for "hash a password with SHA-256", because speed is the enemy there. Choosing a primitive is mostly choosing which row your problem lives in.

Two limits frame everything that follows. First, the Ling-specific layers (the "geometric" hash, the DICE-42 record cipher inside `lingtp://`, the LRS-87 ring signature) are original designs with no third-party audit in the repository. Use the standard primitives for anything that guards real value. Second, all of this protects data at rest and in transit. It does nothing for a machine that is already compromised, a keylogger, or a user who pastes the passphrase into a chat window.

Hashing: fingerprints, never secrets

A hash turns any input into a fixed-size digest. Change one character and the digest changes completely. Ling offers several, and the names are a trap if you do not know what each does.

```
bind start = do {
  bind a = "pay alice 100"
  bind b = "pay alice 101"
  println("sha256   ", sha256_hex(a))
  println("sha256   ", sha256_hex(b))
  println("blake3   ", hash_hex(a))
  println("holo     ", crypto_hash(a))
  println("same?    ", sha256_hex(a) == sha256_hex(a))
  println("bits len ", len(sha256_hex(a)) * 4)
  bind digest = shake_hex(a, 8)
  println("shake    ", digest)
}
```

```
OUTPUT
sha256   4fd0e8ada694a40660c8e8249f444a9e5af14427254b579f419793d6b35f8e18
sha256   605805cb9986570e920d3bfd3a882c91805745b384902b0f29ad31c3ae43b63c
blake3   f788e939d3aca564d2904e8196ab377b23991b093eeace98a5a5c5e386eea271
holo     7b60479c4ef9323718a71f9365dabcaa884cdc49b73ed2f21dd499f3e309d85c
same?    true
bits len 256
shake    c50dfa3e4b635bc1
```

`sha256_hex` is plain SHA-256 and matches what a browser or `sha256sum` prints; use it to interoperate. `hash_hex` is BLAKE3, faster and the better default when both ends are yours. `crypto_hash` is **not** plain SHA3-256: it prepends the label `ling-geo-holo-hash-v1`, so no other tool will reproduce it. `shake_hex(text, n)` squeezes `n` bytes out of SHAKE-256 when you need a digest of unusual length.

For files, use `file_hash`. It hashes raw bytes, so unlike `read_file` it is safe for binaries, and it returns an empty string rather than an error when the file is missing, so test for that.

```
bind start = do {
  write_file("release.txt", "ling-app 1.4.2 build 77")
  bind expected = hash_hex("ling-app 1.4.2 build 77")
  println("file_hash matches text hash: ", file_hash("release.txt") == expected)
  write_file("release.txt", "ling-app 1.4.2 build 78")
  println("after tamper: ", file_hash("release.txt") == expected)
  println("missing file: [", file_hash("nope.bin"), "]")
  remove_file("release.txt")
}
```

```
OUTPUT
file_hash matches text hash: true
after tamper: false
missing file: []
```

WARNING

A hash proves nothing about who produced the data. If an attacker can replace both the file and the digest sitting next to it, the check passes. To bind integrity to an identity, sign the digest (below) or authenticate it with a secret key.

> **NOTE**
>
> `hash_str` and `hash_int` are FNV-1a hashes for bucketing and procedural content. They are fast and trivially forgeable. Never feed them anything an attacker influences and then trust the result.

Sealing: authenticated encryption

`crypto_seal(key, text)` encrypts and authenticates in one step with XChaCha20-Poly1305. The result is the hex of `nonce (24 bytes) || ciphertext || tag (16 bytes)`, so a message of *n* bytes becomes `2 * (40 + n)` hex characters. A fresh random nonce is drawn on every call, which is why sealing the same text twice gives different output and why you never manage nonces yourself.

```
bind start = do {
  bind key = random_hex(32)
  bind ct = crypto_seal(key, "meet at the bridge")
  println("key bytes   ", len(key) / 2)
  println("ct hex len  ", len(ct), " = 2*(40+18)")
  println("again same? ", ct == crypto_seal(key, "meet at the bridge"))
  println(crypto_open(key, ct))
  bind first = substr(ct, 0, 1)
  bind swap = "0"
  if first == "0" { bind swap = "1" }
  bind flipped = swap + substr(ct, 1, len(ct) - 1)
  println("bit flip    ", crypto_open(key, flipped))
  println("wrong key   ", crypto_open(random_hex(32), ct))
  println("truncated   ", crypto_open(key, "abcd"))
}
```

```
OUTPUT
key bytes   32
ct hex len  116 = 2*(40+18)
again same? false
meet at the bridge
bit flip    Err(decryption/auth failed)
wrong key   Err(decryption/auth failed)
truncated   Err(too short)
```

Authentication is the point. A flipped bit, a wrong key and a truncated blob all fail with an `Err` instead of returning garbage. Always handle that result; a `match` on `ok(v)` and `bad(e)` is the idiom.

Two properties to remember. The plaintext is treated as UTF-8 text, so binary data does not round-trip; base64-encode it first. And the ciphertext leaks the length of the message. If length is sensitive (a yes/no flag, say), pad the plaintext to a fixed size before sealing.

The key is the whole game

Here is the footgun the reference chapter warns about, shown live. The hex-decoding step behind every keyed builtin is lenient: a short key is zero-padded, a long one is truncated, and any pair of characters that is not valid hex is silently dropped.

```
bind start = do {
  bind padded = "ab" + str_repeat("00", 31)
  bind ct = crypto_seal("ab", "top secret")
  println("2-char key opens with its zero-padded twin: ", crypto_open(padded, ct))
  bind ct2 = crypto_seal("not hex at all!", "also secret")
  println("garbage key opens with the all-zero key:    ", crypto_open(str_repeat("00",
32), ct2))
  bind typo = str_repeat("11", 10) + "1g" + str_repeat("11", 21)
  bind ct3 = crypto_seal(typo, "typo secret")
  println("typo key == real key? ", crypto_open(str_repeat("11", 32), ct3))
  println("typo key == key with one dropped byte, zero tail: ",
crypto_open(str_repeat("11", 31) + "00", ct3))
}
```

```
OUTPUT
2-char key opens with its zero-padded twin: top secret
garbage key opens with the all-zero key:    also secret
typo key == real key? Err(decryption/auth failed)
typo key == key with one dropped byte, zero tail: typo secret
```

Read that output slowly. A two-character "key" is the same key as `ab` followed by thirty-one zero bytes. A key made of English words is the all-zero key. And a single typo (`1g` instead of `11`) does not raise an error; it produces a different, still weak key. Seeded builtins such as `ed25519_keygen_from_seed` decode their argument the same way.

The defence is a gatekeeper that you call before any key reaches a cipher. It costs a dozen lines:

```
fn is_hex(s) {
  bind i = 0
  while i < len(s) {
    if str_find("0123456789abcdef", substr(s, i, 1)) < 0 { return false }
    bind i = i + 1
  }
  return true
}

fn strong_key(k) {
  return len(k) == 64 && is_hex(k) && k != str_repeat("0", 64)
}

fn must_key(k) {
  if !strong_key(k) { return bad("refusing weak or malformed key") }
  return ok(k)
}

bind start = do {
  println(strong_key(random_hex(32)))
  println(strong_key("ab"))
  println(strong_key(str_repeat("0", 64)))
  println(strong_key(str_repeat("z", 64)))
  println(must_key("abcd"))
  println(must_key(str_repeat("7f", 32)))
}
```

```
OUTPUT
true
false
false
false
Err(refusing weak or malformed key)
Ok(7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f)
```

> **TIP**
>
> Generate keys with `random_hex(32)`, which draws from the operating-system CSPRNG. Never use `rand()`; it is the game random number generator, unseeded and unsuitable for secrets.

> **WARNING**
>
> `==` on strings is not constant time. For comparing secrets (MACs, tokens) prefer the verification builtins, which are built for it. Use `==` only on public values.

Keys from passphrases

People remember passphrases, not 32 random bytes. `lw_password_key(passphrase, salt_hex, m_kib, t, p)` turns one into a key with Argon2id, a deliberately slow, memory-hungry function. It returns `Ok(key_hex)` or `Err(...)`, and it is strict: a bad parameter is an error, unlike the lenient key decoder above.

Three rules make the derivation sound. Use a fresh random 16-byte salt per vault and store it beside the ciphertext; it need not be secret. Keep the parameters in your file format so you can raise them later. And never use the passphrase-derived key for more than one job: derive purpose-specific subkeys with `lw_h256(label, [key])`.

```
fn u(r) { return match r { ok(v) => v, bad(e) => "ERR:" + e } }

bind start = do {
  bind salt = random_hex(16)
  bind master = u(lw_password_key("correct horse", salt, 4096, 2, 1))
  bind enc_key = lw_h256("notes.enc.v1", [master])
  bind mac_key = lw_h256("notes.mac.v1", [master])
  bind sign_seed = lw_h256("notes.sign.v1", [master])
  println("master:   ", len(master) / 2, " bytes")
  println("distinct: ", enc_key != mac_key && mac_key != sign_seed && enc_key !=
master)
  bind again = u(lw_password_key("correct horse", salt, 4096, 2, 1))
  println("repeatable: ", again == master)
  bind other = u(lw_password_key("correct horse", random_hex(16), 4096, 2, 1))
  println("new salt differs: ", other != master)
  println("bad params: ", u(lw_password_key("x", salt, 0, 0, 0)))
}
```

```
OUTPUT
master:   32 bytes
distinct: true
repeatable: true
new salt differs: true
bad params: ERR:invalid Argon2id parameters
```

`lw_h256` is domain separated: the same master with a different label yields an unrelated key, and each part is length-prefixed so `["ab","cd"]` differs from `["abcd"]`. That gives you an encryption key, a manifest key and a signing seed from one passphrase, none of which reveals another.

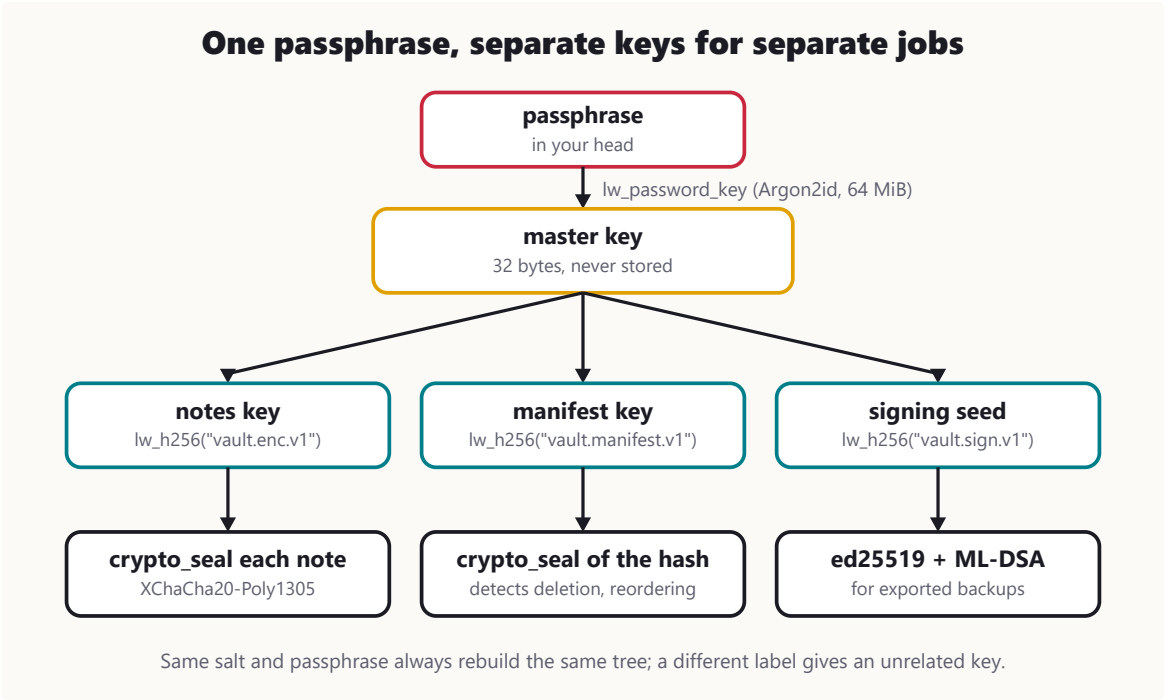


Figure: the key hierarchy used by the vault later in this chapter. The master key is never written to disk.

How slow should the derivation be? The example above uses 4 MiB and two passes so it runs instantly. For a vault, take at least 64 MiB and three passes, the profile we use below; the wallet profile in the reference is 256 MiB. Measure on your slowest target machine and spend the most waiting time your users will tolerate. That cost is the cap on how fast an attacker holding a stolen file can guess.

Signatures: who published this?

A signature proves that the holder of a private key approved a message, and anyone with the public key can check it. Ling has three families with the same shape: key generation returns a handle, `*_public` reads the public half, `*_sign` signs a string, `*_verify` checks it. The private key never enters your program; you hold only a small integer.

```
bind start = do {
  bind ed = ed25519_keygen()
  bind pq = mldsa87_keygen()
  bind doc = "release=ling-app-1.4.2 sha=4fd0e8ad"
  bind s1 = ed25519_sign(ed, doc)
  bind s2 = mldsa87_sign(pq, doc)
  println("ed25519 pk/sig bytes: ", len(ed25519_public(ed)) / 2, " / ", len(s1) / 2)
  println("ml-dsa-87 pk/sig bytes: ", len(mldsa87_public(pq)) / 2, " / ", len(s2) / 2)
  println("both verify: ", ed25519_verify(ed25519_public(ed), doc, s1) &&
mldsa87_verify(mldsa87_public(pq), doc, s2))
  bind forged = doc + "!"
  println("tampered doc, ed25519: ", ed25519_verify(ed25519_public(ed), forged, s1))
  println("tampered doc, ml-dsa:  ", mldsa87_verify(mldsa87_public(pq), forged, s2))
  println("ed25519 deterministic: ", ed25519_sign(ed, doc) == s1)
  println("ml-dsa deterministic:  ", mldsa87_sign(pq, doc) == s2)
  println("garbage public key:    ", ed25519_verify("00", doc, s1))
}
```

```
OUTPUT
ed25519 pk/sig bytes: 32 / 64
ml-dsa-87 pk/sig bytes: 2592 / 4627
both verify: true
tampered doc, ed25519: false
tampered doc, ml-dsa:  false
ed25519 deterministic: true
ml-dsa deterministic:  true
garbage public key:    false
```

Ed25519 keys and signatures are tiny (32 and 64 bytes) and fast, but a future quantum computer can forge them. ML-DSA-87 (FIPS 204) is quantum resistant, at the price of 2,592-byte public keys and 4,627-byte signatures. For artifacts that must stay trustworthy for years (licence files, release manifests, firmware) sign with both and require both. The verify functions return `false` for any malformed input rather than raising, so a garbage public key is simply a failure.

Key storage is where signing systems actually fail. A handle lives only as long as the process. To restore the same identity after a restart, keep a 32-byte seed in protected storage and rebuild with `ed25519_keygen_from_seed`. Do not reuse one seed for both families: derive a

separate seed per algorithm and purpose.

```
fn both_ok(ed_pub, pq_pub, msg, ed_sig, pq_sig) {
  return ed25519_verify(ed_pub, msg, ed_sig) && mldsa87_verify(pq_pub, msg, pq_sig)
}

bind start = do {
  bind seed = random_hex(32)
  bind ed = ed25519_keygen_from_seed(lw_h256("licence.ed25519.v1", [seed]))
  bind pq = mldsa87_keygen_from_seed(lw_h256("licence.mldsa87.v1", [seed]))
  bind ed_pub = ed25519_public(ed)
  bind pq_pub = mldsa87_public(pq)
  bind m = "licence: acme, expires 2031-01-01"
  bind es = ed25519_sign(ed, m)
  bind ps = mldsa87_sign(pq, m)
  println("fresh: ", both_ok(ed_pub, pq_pub, m, es, ps))
  println("classical sig broken: ", both_ok(ed_pub, pq_pub, m, "00" + substr(es, 2,
126), ps))
  println("pq sig missing:       ", both_ok(ed_pub, pq_pub, m, es, ""))
  bind again = ed25519_keygen_from_seed(lw_h256("licence.ed25519.v1", [seed]))
  println("seed restores same public key: ", ed25519_public(again) == ed_pub)
}
```

```
OUTPUT
fresh: true
classical sig broken: false
pq sig missing:       false
seed restores same public key: true
```

TIP

Version your labels (`licence.ed25519.v1`). When you must rotate a key you can introduce `v2` with no chance of the two generations colliding.

NOTE

Both algorithms here are deterministic: the same key and message give the same signature. That removes a whole class of nonce-reuse bugs.

Passwords and second factors

Never store a password, and never store a fast hash of one. `password_hash` produces a self-describing PHC string with a random salt, and `password_verify` checks against it. The parameters are fixed in the runtime (64 MiB, three passes, four lanes), a sensible default for a login endpoint.

```
bind start = do {
  bind stored = password_hash("correct horse battery")
  println("phc prefix: ", substr(stored, 0, 30))
  println("phc length: ", len(stored))
  println("right: ", password_verify("correct horse battery", stored))
  println("wrong: ", password_verify("correct horse batterz", stored))
  println("same password, new salt: ", password_hash("correct horse battery") !=
stored)
  println("garbage record: ", password_verify("x", "not-a-phc-string"))
}
```

```
OUTPUT
phc prefix: $argon2id$v=19$m=65536,t=3,p=4
phc length: 97
right: true
wrong: false
same password, new salt: true
garbage record: false
```

The record is 97 characters; store it in a single text column. It embeds the algorithm, parameters and salt, so old records stay verifiable after defaults change. A malformed record verifies as `false`, never as an error, which makes the check safe to call on whatever the database returns.

A second factor makes a stolen password insufficient. `totp_secret` creates a 160-bit Base32 secret, `totp_uri` builds the `otpauth://` link that authenticator apps import, and `lw_qr` renders it as QR modules you can print or draw.

```
bind start = do {
  bind secret = totp_secret()
  println("secret length: ", len(secret))
  bind uri = totp_uri(secret, "alice@example.com", "Acme")
  println(substr(uri, 0, 38))
  println("params: ", substr(uri, len(uri) - 38, 38))
  bind code = totp_now(secret)
  println("code digits: ", len(code))
  println("accepts current code: ", totp_verify(secret, code))
  println("rejects 000000 (unless unlucky): ", totp_verify(secret, "000000") == (code
== "000000"))
  println("other secret rejects: ", totp_verify(totp_secret(), code) == false)
  bind rows = lw_qr(uri)
  println("QR is ", len(rows), " x ", len(rows[0]), " modules")
}
```

```
OUTPUT
secret length: 32
otpauth://totp/Acme:alice@example.com?
params: Acme&algorithm=SHA1&digits=6&period=30
code digits: 6
accepts current code: true
rejects 000000 (unless unlucky): true
other secret rejects: true
QR is 45 x 45 modules
```

The codes are SHA-1, six digits, thirty seconds, which is what every mainstream authenticator expects. `totp_verify` accepts one step either side of now to forgive clock drift. It does **not** remember what it has accepted, so an attacker who sees a code can replay it within the window. You add that memory yourself:

```
// A second-factor check that also refuses replays.
fn already_used(path, code) {
  if !file_exists(path) { return false }
  bind seen = split(read_file(path), "\n")
  bind i = 0
  while i < len(seen) {
    if seen[i] == code { return true }
    bind i = i + 1
  }
  return false
}

fn check_2fa(secret, code, used_path) {
  if already_used(used_path, code) { return bad("code already used") }
  if !totp_verify(secret, code) { return bad("wrong code") }
  bind prev = ""
  if file_exists(used_path) { bind prev = read_file(used_path) + "\n" }
  write_file(used_path, prev + code)
  return ok("welcome")
}

bind start = do {
  bind secret = totp_secret()
  bind code = totp_now(secret)
  println("first use:  ", check_2fa(secret, code, "used_codes.txt"))
  println("replay:     ", check_2fa(secret, code, "used_codes.txt"))
  println("bad code:   ", check_2fa(secret, "123456", "used_codes.txt"))
  remove_file("used_codes.txt")
}
```

```
OUTPUT
first use:  Ok(welcome)
replay:     Err(code already used)
bad code:   Err(wrong code)
```

> **WARNING**
>
> The replay file above is a teaching device. In a real service, store the last accepted code per user in your database, expire entries after about ninety seconds, and rate-limit attempts per account. A six-digit code has only a million values.

Post-quantum key agreement

To share a secret with someone over a channel an eavesdropper can read, use a key encapsulation mechanism. The recipient publishes a public key; the sender "encapsulates" a fresh shared secret to it and sends back a ciphertext; only the recipient's private key recovers the same secret. Ling's `knot_*` family combines X25519 and ML-KEM-768, so the result stays safe if either half is broken.

```
fn u(r) { return match r { ok(v) => v, bad(e) => "ERR:" + e } }

bind start = do {
  bind bob = knot_keygen()
  bind enc = knot_encapsulate(knot_public(bob))
  println("public key bytes: ", len(knot_public(bob)) / 2)
  println("ciphertext bytes: ", len(enc[0]) / 2)
  println("secret length: ", len(enc[1]))
  bind same = knot_decapsulate(bob, enc[0])
  println("bob recovers secret: ", same == enc[1])
  bind sealed = crypto_seal(enc[1], "hello bob, this is alice")
  println(crypto_open(same, sealed))
  bind tampered = "00" + substr(enc[0], 2, len(enc[0]) - 2)
  bind wrong = knot_decapsulate(bob, tampered)
  println("tampered ct yields a different secret: ", wrong != enc[1])
  println("and then fails authentication: ", crypto_open(wrong, sealed))
}
```

```
OUTPUT
public key bytes: 1216
ciphertext bytes: 1120
secret length: 64
bob recovers secret: true
hello bob, this is alice
tampered ct yields a different secret: true
and then fails authentication: Err(decryption/auth failed)
```

The shared secret is 32 bytes (64 hex characters), exactly the shape `crypto_seal` wants as a key. Look at the last two lines: ML-KEM uses implicit rejection. A tampered ciphertext does not produce an error; it produces a different, random-looking secret. Detection therefore happens one step later, when authentication under that secret fails. Always use an agreed secret to authenticate something before you trust it.

> **NOTE**
>
> `knot_keygen` has no seed and no export, so its private key lives and dies with the process. For a persistent identity use `lw_kem_keypair`, which takes a 96-byte seed, or the `HybridKeypair` API in Rust. Inside a restricted isolate, the key-holding `lw_*` calls need the `"keys"` capability.

Mini-project: an encrypted notes vault

Time to combine the pieces. The vault stores notes in a single file. Requirements, stated as a threat model:

- **Assets:** note titles and bodies.
- **Attacker:** can read, copy and modify the file; has unlimited offline time but not the passphrase.
- **Must hold:** confidentiality of every title and body; a wrong passphrase is detected cleanly; modification, deletion or reordering of notes is detected.
- **Out of scope:** malware on the machine, shoulder surfing, and rollback to an older valid copy.

The design follows. One Argon2id derivation yields a master key, from which `lw_h256` derives an encryption key and a manifest key. Each note is sealed individually, title and body together. A sealed check line lets us reject a wrong passphrase before touching any note. A

final manifest line seals the hash of all note lines, so removing or reordering one breaks it. The file is plain text, one record per line:

```
OUTPUT
LVT1
<salt hex>
<sealed "vault-ok">
<sealed note> ...
<sealed hash of all note lines>
```

```
// vault.ling — an encrypted notes vault.
// File layout (one record per line): LVT1 / salt / sealed check / sealed notes... /
sealed manifest

fn u(r) { return match r { ok(v) => v, bad(e) => "ERR:" + e } }

fn derive(pass, salt) {
  bind master = u(lw_password_key(pass, salt, 65536, 3, 1))
  return [lw_h256("vault.enc.v1", [master]), lw_h256("vault.manifest.v1", [master])]
}

fn manifest(mkey, notes) {
  return crypto_seal(mkey, hash_hex(list_join(notes, "\n")))
}

fn save(path, salt, keys, notes) {
  bind lines = ["LVT1", salt, crypto_seal(keys[0], "vault-ok")]
  bind i = 0
  while i < len(notes) {
    bind lines = list_push(lines, notes[i])
    bind i = i + 1
  }
  bind lines = list_push(lines, manifest(keys[1], notes))
  write_file(path, list_join(lines, "\n"))
}

// Returns ok([salt, keys, notes]) or bad(reason).
fn open_vault(path, pass) {
  if !file_exists(path) { return bad("no such vault") }
  bind lines = split(read_file(path), "\n")
  if len(lines) < 4 || lines[0] != "LVT1" { return bad("not a vault file") }
  bind salt = lines[1]
  bind keys = derive(pass, salt)
  if crypto_open(keys[0], lines[2]) != "vault-ok" { return bad("wrong passphrase or
damaged header") }
  bind notes = []
  bind i = 3
  while i < len(lines) - 1 {
```

```
    bind notes = list_push(notes, lines[i])
    bind i = i + 1
  }
  if crypto_open(keys[1], lines[len(lines) - 1]) != hash_hex(list_join(notes, "\n")) {
    return bad("manifest mismatch: a note was removed, added or reordered")
  }
  return ok([salt, keys, notes])
}

fn vault_new(path, pass) {
  bind salt = random_hex(16)
  save(path, salt, derive(pass, salt), [])
}

fn add_note(path, pass, title, body) {
  return match open_vault(path, pass) {
    ok(v) => do {
      bind notes = list_push(v[2], crypto_seal(v[1][0], title + "\n" + body))
      save(path, v[0], v[1], notes)
      ok(len(notes))
    },
    bad(e) => bad(e)
  }
}

fn read_note(path, pass, title) {
  return match open_vault(path, pass) {
    ok(v) => do {
      bind found = bad("no note titled " + title)
      bind i = 0
      while i < len(v[2]) {
        bind plain = crypto_open(v[1][0], v[2][i])
        bind cut = str_find(plain, "\n")
        if substr(plain, 0, cut) == title { bind found = ok(substr(plain, cut + 1,
len(plain) - cut - 1)) }
        bind i = i + 1
      }
      found
    },
```

```
    bad(e) => bad(e)
  }
}

fn titles(path, pass) {
  return match open_vault(path, pass) {
    ok(v) => do {
      bind out = []
      bind i = 0
      while i < len(v[2]) {
        bind plain = crypto_open(v[1][0], v[2][i])
        bind out = list_push(out, substr(plain, 0, str_find(plain, "\n")))
        bind i = i + 1
      }
      ok(out)
    },
    bad(e) => bad(e)
  }
}

bind start = do {
  bind path = "demo.vault"
  vault_new(path, "tr0ub4dor&3 horse")
  println(add_note(path, "tr0ub4dor&3 horse", "bank", "pin is 4921"))
  println(add_note(path, "tr0ub4dor&3 horse", "wifi", "guest network: sunflower-88"))
  println(titles(path, "tr0ub4dor&3 horse"))
  println(read_note(path, "tr0ub4dor&3 horse", "wifi"))
  println(read_note(path, "tr0ub4dor&3 horse", "missing"))
  println(read_note(path, "wrong passphrase", "wifi"))
  println("plaintext on disk? ", read_file(path).contains("sunflower"))

  // an attacker deletes the first note line
  bind lines = split(read_file(path), "\n")
  write_file("cut.vault", list_join([lines[0], lines[1], lines[2], lines[4], lines[5]],
"\n"))
  println(titles("cut.vault", "tr0ub4dor&3 horse"))
```

```
  remove_file(path)
  remove_file("cut.vault")
}
```

```
OUTPUT
Ok(1)
Ok(2)
Ok([bank, wifi])
Ok(guest network: sunflower-88)
Err(no note titled missing)
Err(wrong passphrase or damaged header)
plaintext on disk? false
Err(manifest mismatch: a note was removed, added or reordered)
```

Trace the output. The wrong passphrase is stopped at the check line. The word "sunflower" appears nowhere on disk. And when the attacker deletes the first note line the file still looks valid (every remaining line decrypts), yet the manifest no longer matches, so the vault refuses to open. Per-note encryption alone cannot give you that last property.

Some honest limits. Every call re-derives the key (a fraction of a second each, which is why the whole run takes several seconds), because the program holds no state between calls; a real tool would open once and keep the keys. The secrets are ordinary strings and are not wiped from memory. An attacker who swaps in an older valid vault defeats the manifest, because it has no counter; that needs state outside the file. And the number and approximate size of notes still leak.

> TRY IT
>
> Add a `vault_count(path, pass)` function, then store a note whose title contains a newline. What happens when you read it back, and how would you reject it at write time?

The same primitives from Rust

Everything above sits on the `ling-crypto` crate, and the formats are the real ones, not Ling-private wrappers. A Rust service and a Ling script can share files. First, print what the builtins produce for fixed inputs:

```
fn u(r) { return match r { ok(v) => v, bad(e) => "ERR:" + e } }

bind start = do {
  println("argon2id  ", u(lw_password_key("hunter2",
"00112233445566778899aabbccddeeff", 1024, 1, 1)))
  println("lw_h256   ", lw_h256("app.v1", ["616263"]))
  println("blake3    ", hash_hex("abc"))
}
```

```
OUTPUT
argon2id  a97cf9fed0c0703d0b60b7351990a06b086636d9fec2c5ef4d2b7a4bae25bb34
lw_h256   b3591f3c59e2a95a244cefed9cb3db5999634fe8945e04a220574d362da63bd7
blake3    6437b3ac38465133ffb63b75273a8db548c558465d79db03fd359c6cd5bd9d85
```

Now have Ling write a key and a sealed message:

```
bind start = do {
  bind key = random_hex(32)
  bind sealed = crypto_seal(key, "release key rotated at 02:00")
  write_file("from_ling.txt", key + "\n" + sealed)
  println("wrote from_ling.txt (", len(sealed) / 2, " sealed bytes)")
}
```

```
OUTPUT
wrote from_ling.txt (68 sealed bytes)
```

A Rust program using `ling_crypto::lw::kdf` and `XChaCha20` reproduces the first three values digit for digit, opens that file, and answers with a message sealed in Rust:

```rust
use ling_crypto::lw::hash::h256;
use ling_crypto::lw::kdf::{password_key, PasswordParams};
use ling_crypto::{Blake3, XChaCha20};

fn hex(b: &[u8]) -> String {
    b.iter().map(|x| format!("{x:02x}")).collect()
}

fn unhex(s: &str) -> Vec<u8> {
    (0..s.len() / 2)
        .map(|i| u8::from_str_radix(&s[2 * i..2 * i + 2], 16).expect("hex"))
        .collect()
}

fn main() {
    // 1. The same Argon2id parameters as lw_password_key("hunter2", salt, 1024, 1, 1).
    let salt: [u8; 16] = unhex("00112233445566778899aabbccddeeff").try_into().unwrap();
    let params = PasswordParams { m_cost_kib: 1024, t_cost: 1, p_cost: 1 };
    let key = password_key(b"hunter2", &salt, params).unwrap();
    println!("argon2id  {}", hex(&*key));

    // 2. lw_h256("app.v1", ["616263"]) takes hex parts; here the parts are raw bytes.
    println!("lw_h256   {}", hex(&h256("app.v1", &[b"abc"])));
    println!("blake3    {}", hex(&Blake3::hash(b"abc")));

    // 3. Open what Ling's crypto_seal wrote: <key hex> newline <sealed hex>.
    let text = std::fs::read_to_string("from_ling.txt").expect("run the Ling half
first");
    let mut lines = text.lines();
    let key: [u8; 32] = unhex(lines.next().unwrap()).try_into().unwrap();
    let sealed = unhex(lines.next().unwrap());
    let cipher = XChaCha20::new(key);
    let plain = cipher.decrypt(&sealed).expect("authentic");
    println!("rust opened: {}", String::from_utf8(plain).unwrap());

    // 4. Seal in Rust; Ling's crypto_open reads the same layout (nonce || ct || tag).
    let reply = cipher.encrypt(b"ack from rust").unwrap();
```

```
    std::fs::write("from_rust.txt", hex(&reply)).unwrap();
    println!("rust sealed {} bytes", reply.len());
}
```

```
OUTPUT
argon2id  a97cf9fed0c0703d0b60b7351990a06b086636d9fec2c5ef4d2b7a4bae25bb34
lw_h256   b3591f3c59e2a95a244cefed9cb3db5999634fe8945e04a220574d362da63bd7
blake3    6437b3ac38465133ffb63b75273a8db548c558465d79db03fd359c6cd5bd9d85
rust opened: release key rotated at 02:00
rust sealed 53 bytes
```

Back in Ling, the Rust reply opens with `crypto_open`:

```
bind start = do {
  bind key = split(read_file("from_ling.txt"), "\n")[0]
  println(crypto_open(key, read_file("from_rust.txt")))
}
```

```
OUTPUT
ack from rust
```

Rust also exposes what the builtins do not. Below, a master key is split with Shamir secret sharing so any three of five guardians can rebuild it, then wrapped for a recipient with the hybrid KEM, then signed with both signature families.

```rust
use ling_crypto::{
    hybrid_encapsulate, reconstruct_secret, split_secret, Ed25519Keypair,
HybridKeypair,
    MlDsa87Keypair, XChaCha20,
};

fn main() {
    // A vault master key, split so that any 3 of 5 guardians can rebuild it.
    let master = XChaCha20::generate_key();
    let shares = split_secret(&master, 3, 5);
    println!("shares: {}, each {} bytes (x = {:?})", shares.len(), shares[0].y.len(),
        shares.iter().map(|s| s.x).collect::<Vec<_>>());

    let back = reconstruct_secret(&[shares[4].clone(), shares[0].clone(),
shares[2].clone()]);
    println!("3 shares rebuild the key: {}", back == master);
    let short = reconstruct_secret(&[shares[1].clone(), shares[3].clone()]);
    println!("2 shares rebuild the key: {}", short == master);

    // Wrap the master key for a recipient with the hybrid X25519 + ML-KEM-768 KEM.
    let recipient = HybridKeypair::generate();
    let (kem_ct, shared) = hybrid_encapsulate(&recipient.public_key()).unwrap();
    let wrap = XChaCha20::new(shared_to_key(&shared));
    let wrapped = wrap.encrypt(&master).unwrap();
    println!("kem ciphertext {} bytes, wrapped key {} bytes", kem_ct.len(),
wrapped.len());

    let shared2 = recipient.decapsulate(&kem_ct).unwrap();
    let unwrapped = XChaCha20::new(shared_to_key(&shared2)).decrypt(&wrapped).unwrap();
    println!("recipient recovers key: {}", unwrapped == master);

    // Sign the wrapped blob with both signature families; accept only if both verify.
    let ed = Ed25519Keypair::generate();
    let pq = MlDsa87Keypair::generate();
    let (s1, s2) = (ed.sign(&wrapped), pq.sign(&wrapped));
    let ok = Ed25519Keypair::verify(&ed.public_key(), &wrapped, &s1).is_ok()
        && MlDsa87Keypair::verify(&pq.public_key(), &wrapped, &s2).is_ok();
    println!("hybrid signature verifies: {ok}");
```

```
    let mut bad = wrapped.clone();
    bad[0] ^= 1;
    println!("tampered blob verifies: {}",
        Ed25519Keypair::verify(&ed.public_key(), &bad, &s1).is_ok());
}

fn shared_to_key(shared: &[u8]) -> [u8; 32] {
    let mut k = [0u8; 32];
    k.copy_from_slice(&shared[..32]);
    k
}
```

OUTPUT

```
shares: 5, each 32 bytes (x = [1, 2, 3, 4, 5])
3 shares rebuild the key: true
2 shares rebuild the key: false
kem ciphertext 1120 bytes, wrapped key 72 bytes
recipient recovers key: true
hybrid signature verifies: true
tampered blob verifies: false
```

TIP

`split_secret` and `reconstruct_secret` panic on invalid arguments (a threshold below 2, or an empty secret). Validate input at your API boundary. And note the quiet failure mode in the output: two shares of a three-share scheme "reconstruct" a wrong value rather than raising an error, so confirm any reconstruction by using it, for example by opening a sealed check value.

Chapter summary

- Start from a threat model: assets, attacker, position. Pick the primitive from that, not from the function list.
- `hash_hex` (BLAKE3) and `sha256_hex` fingerprint data; `crypto_hash` is label-prefixed SHA3 and interoperates with nothing. No hash authenticates by itself.
- `crypto_seal` / `crypto_open` give authenticated encryption with a fresh nonce. Handle the `Err`.

- Keyed builtins decode hex leniently: short keys are zero-padded and typos are dropped. Validate every key with a gatekeeper and generate keys with `random_hex(32)`.
- Derive keys from passphrases with `lw_password_key` and a random salt, then split them into purpose-specific subkeys with `lw_h256`.
- Sign long-lived artifacts with Ed25519 and ML-DSA-87 together; seeds restore identities, and each algorithm gets its own derived seed.
- Use `password_hash` for logins and TOTP for a second factor, with your own replay guard.
- KEM secrets must be used to authenticate something; tampering yields a different secret, not an error.
- The Ling-specific constructions are unaudited. The Rust crate and the builtins share formats, so the two worlds interoperate.

Exercises

1. **A safe seal.** Write `seal_checked(key, text)` that returns `bad(...)` for a key that is not exactly 64 lowercase hex characters or is all zeros, and `ok(ciphertext)` otherwise. Show it rejecting `"ab"`.
2. **Two signatures, one bundle.** Write `sign_bundle` and `verify_bundle` so a release carries one string `"<ed25519>:<ml-dsa-87>"` that verifies only if both halves verify. Show that a different message and a malformed bundle both fail.
3. **Threat review.** For the vault above, say whether each attacker succeeds and why: (a) steals the file and has a laptop for a month; (b) deletes the last note; (c) replaces the file with last week's backup; (d) installs a keylogger.

Answers

1. The gatekeeper from earlier, wrapped in a result so callers cannot ignore it:

```
fn is_hex(s) {
  bind i = 0
  while i < len(s) {
    if str_find("0123456789abcdef", substr(s, i, 1)) < 0 { return false }
    bind i = i + 1
  }
  return true
}

fn seal_checked(key, text) {
  if len(key) != 64 || !is_hex(key) { return bad("key must be 64 lowercase hex
characters") }
  if key == str_repeat("0", 64) { return bad("all-zero key refused") }
  return ok(crypto_seal(key, text))
}

bind start = do {
  println(seal_checked("ab", "x"))
  println(seal_checked(str_repeat("0", 64), "x"))
  println(seal_checked("ABCD" + str_repeat("ef", 30), "x"))
  match seal_checked(random_hex(32), "x") {
    ok(c) => println("sealed ", len(c) / 2, " bytes"),
    bad(e) => println(e)
  }
}
```

```
OUTPUT
Err(key must be 64 lowercase hex characters)
Err(all-zero key refused)
Err(key must be 64 lowercase hex characters)
sealed 41 bytes
```

2. Split on the colon and require both checks. The malformed inputs return `false` because the verify functions never raise:

```
// A release bundle: "<ed25519 sig>:<ml-dsa-87 sig>", valid only when both verify.
fn sign_bundle(ed, pq, msg) {
  return ed25519_sign(ed, msg) + ":" + mldsa87_sign(pq, msg)
}

fn verify_bundle(ed_pub, pq_pub, msg, bundle) {
  bind parts = split(bundle, ":")
  if len(parts) != 2 { return false }
  return ed25519_verify(ed_pub, msg, parts[0]) && mldsa87_verify(pq_pub, msg, parts[1])
}

bind start = do {
  bind ed = ed25519_keygen()
  bind pq = mldsa87_keygen()
  bind b = sign_bundle(ed, pq, "v1.4.2")
  println("bundle length: ", len(b))
  println("good: ", verify_bundle(ed25519_public(ed), mldsa87_public(pq), "v1.4.2", b))
  println("other message: ", verify_bundle(ed25519_public(ed), mldsa87_public(pq),
"v1.4.3", b))
  println("half a bundle: ", verify_bundle(ed25519_public(ed), mldsa87_public(pq),
"v1.4.2", "00:11"))
  println("no colon: ", verify_bundle(ed25519_public(ed), mldsa87_public(pq), "v1.4.2",
"nothing"))
}
```

```
OUTPUT
bundle length: 9383
good: true
other message: false
half a bundle: false
no colon: false
```

3. (a) Fails if the passphrase is strong: each guess costs a 64 MiB Argon2id run, so a month of guessing covers only an unlucky weak passphrase. A short common passphrase would fall, which is why passphrase quality matters more than any cost parameter. (b) Detected. Deleting the last note changes the hash of the note lines, so the sealed manifest no longer matches. (c) Succeeds. The old file is internally consistent and the vault has no counter or

external clock. Detecting it needs state outside the file, such as a monotonic counter kept by a server or a signed timestamp. (d) Succeeds. A keylogger captures the passphrase before any cryptography runs; no file format helps once the attacker is on the machine.

CHAPTER 6

Peer-to-Peer and Protocols

What you'll learn

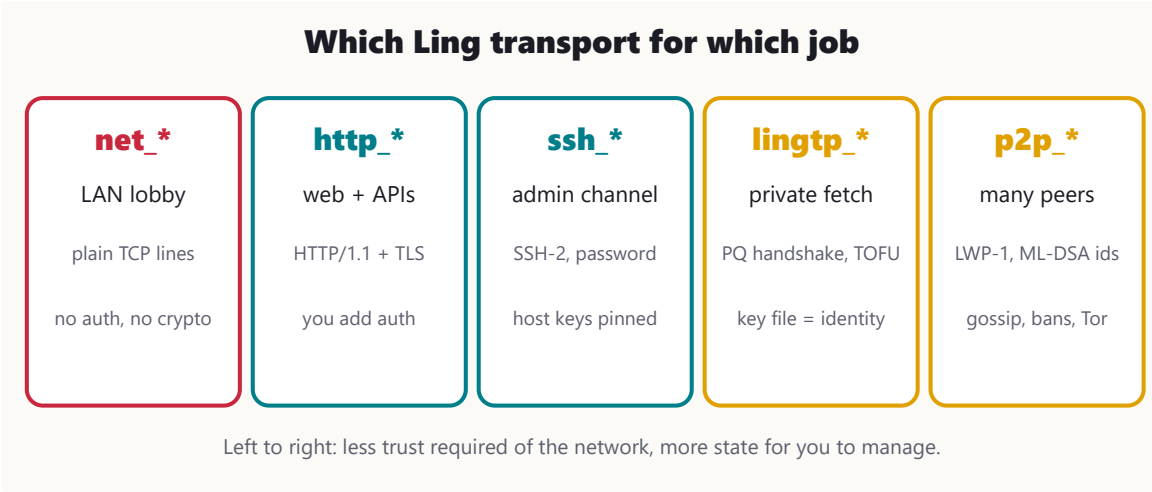
HTTP is the default answer, but it is not the only one, and Ling ships four other ways to move bytes between programs: a plain TCP lobby (`net_*`), a post-quantum request/response protocol (`lingtp_*`), an SSH server and client (`ssh_*`), and a peer-to-peer transport (`p2p_*`). Each exists for a reason, each has sharp edges, and several have never been exercised by anyone but their authors. This chapter is the honest tour: what each transport is for, how to design a protocol on top of it, which parts we ran and which we could not, and how to interoperate with a Rust program using the `ling-net` crate.

You will:

- pick a transport from a table of what it guarantees and what it leaves to you;
- design and run a line-framed protocol over `net_*`, including the framing mistake that lets one client forge another's messages;
- run a key-pinned `lingtp` service, watch the client refuse an impostor, and learn why the Rust and Ling implementations cannot talk to each other;
- connect a Rust node to a Ling `p2p_*` node and exchange messages;
- build a three-node gossip network where a message crosses a node that relays it;
- read a verified-versus-not table so that you know exactly where our testing stopped.

You need the earlier chapters on loops and records, and enough Rust to read one 100-line program. A second terminal is useful, since several examples run a server in one process and a client in another. Everything was run on one Windows machine over loopback with `ling` 2030.1.53; that matters, because loopback hides every problem that depends on a real network (NAT, packet loss, latency, firewalls). Treat our results as proof that the *code paths work*, not that your deployment will.

Choosing a transport



The five transports from left to right: less trust required of the network, more state for you to manage.

Transport	Wire	Identity and trust	Good for	Leaves to you
`net_*`	One UTF-8 line per message over TCP	None. No authentication, no encryption	LAN games, local tools, prototypes	Everything: auth, framing rules, limits
`http_*`	HTTP/1.1, optional TLS	Certificates (you provide them)	APIs, browsers, anything with a proxy in front	Auth, sessions, concurrency design
`ssh_*`	SSH-2, password authentication only	Host key pinned on first use	An admin or command channel to existing OpenSSH hosts	Command semantics, key-based login (absent)
`lingtp_*`	Post-quantum handshake, sealed frames, no certificate authority	Server key pinned on first use; the key file is the identity	Private service-to-service calls where you control both ends	Application protocol, key backup, one implementation on both ends
`p2p_*`	LWP-1 with ML-DSA-87 identities and hybrid key exchange	Per-peer authenticated identities, optional pinned seed list	Many equal nodes, gossip, relays	Message meaning, peer policy, discovery

Two rules drive the choice. First, if the peers are not all under your control, use HTTP with TLS: browsers, proxies and monitoring tools only understand it. Second, use the exotic transports when you control both ends and want properties HTTP lacks: no certificate authority (`lingtp`), no server at all (`p2p`).

> **WARNING**
>
> Every transport in this chapter except `http_*` with TLS is young. The post-quantum cryptography is real (it is implemented in the `ling-crypto` crate and uses standard algorithm families), but neither the protocols nor the code have had independent review. For anything that protects money or health, use a reviewed stack such as TLS or OpenSSH, and use these for experiments, game traffic and internal tooling.

Framing: designing a protocol over `net_*`

`net_*` gives you a reliable, ordered stream of text lines and nothing else. Everything above that, the *protocol*, is yours. Even a trivial one needs decisions about framing (where does a message end?), verbs (what can a peer say?), errors (what does a peer see when it says nonsense?) and limits (how big?). Here is a small chat room that makes those decisions explicitly.

The wire format is one line per frame, `VERB|argument`. The server understands `HELLO|name`, `SAY|text` and `BYE|`; it answers with `MSG|who|text` frames to everyone and `ERR|reason` frames to the offender. The host keeps a name table keyed by the client id that the runtime reports (`net_recv_from` returns `"id|payload"`, so the sender is established by the transport and cannot be forged by the payload).

GUI

```
// One program, two roles:  ling run chat.ling --role host
//                          ling run chat.ling --role client --name ada
// Wire format: one line per frame,  VERB|argument
fn verb_of(line) {
  bind at = str_find(line, "|")
  if at < 0 { return line }
  return substr(line, 0, at)
}

fn arg_of(line) {
  bind at = str_find(line, "|")
  if at < 0 { return "" }
  return substr(line, at + 1, len(line))
}

// "id|payload" from net_recv_from -> payload
fn payload_of(line) { return arg_of(line) }
fn sender_of(line) { return to_number(verb_of(line)) }

fn name_of(table, id) {
  bind key = ";" + id + "="
  bind at = str_find(table, key)
  if at < 0 { return "?" }
  bind rest = substr(table, at + len(key), len(table))
  return substr(rest, 0, str_find(rest, ";"))
}

fn run_host() {
  net_host(18130)
  bind table = ";"
  bind quiet = 0
  bind bye = 0
  while quiet < 60 && bye < 2 {
    bind raw = net_recv_from()
    if raw == "" {
      sleep(50)
      bind quiet = quiet + 1
```

```
    } else {
      bind quiet = 0
      bind id = sender_of(raw)
      bind frame = payload_of(raw)
      bind v = verb_of(frame)
      if v == "HELLO" {
        bind table = table + id + "=" + arg_of(frame) + ";"
        net_send("MSG|server|" + arg_of(frame) + " joined")
      } else if v == "SAY" {
        if len(arg_of(frame)) > 200 {
          net_send_to(id, "ERR|message too long")
        } else {
          net_send("MSG|" + name_of(table, id) + "|" + arg_of(frame))
        }
      } else if v == "BYE" {
        bind bye = bye + 1
        net_send("MSG|server|" + name_of(table, id) + " left")
      } else {
        net_send_to(id, "ERR|unknown verb " + v)
      }
    }
  }
  sleep(300)
  println("host done, events: ", net_events())
  net_close()
}

fn run_client(name) {
  net_join("127.0.0.1", 18130)
  bind n = 0
  while net_status() != 2 && n < 100 { sleep(50) bind n = n + 1 }
  net_send("HELLO|" + name)
  sleep(400)
  net_send("SAY|hello, I am " + name)
  net_send("DANCE|now")
  sleep(600)
  net_send("BYE|")
  bind quiet = 0
  while quiet < 12 {
```

```
    bind m = net_recv()
    if m == "" { sleep(50) bind quiet = quiet + 1 } else { println(name, " <- ", m)
bind quiet = 0 }
  }
  net_close()
}

bind start = do {
  if cli_arg("role", "client") == "host" { run_host() } else {
run_client(cli_arg("name", "anon")) }
}
```

Start the host in one terminal and two clients in others (`ling run chat.ling --role host`, then `ling run chat.ling --name ada` and `... --name bob`). Each client joins, says hello, sends one valid message and one invalid verb, then leaves. Here are the host's summary and Bob's view:

```
OUTPUT
host done, events: connect:1
connect:2
```

```
OUTPUT
bob <- MSG|server|bob joined
bob <- MSG|ada|hello, I am ada
bob <- MSG|bob|hello, I am bob
bob <- ERR|unknown verb DANCE
bob <- MSG|server|ada left
bob <- MSG|server|bob left
```

Bob's view shows the design working. He saw his own `joined` frame but not the earlier part of Ada's join (he connected after it), he saw both chat lines, and the `ERR` was sent to him alone because the host used `net_send_to` for it. Ada's transcript differs only in that she saw both `joined` frames, because she connected first. We ran the whole thing under `--interp` as well and the transcripts matched line for line.

Study the helper functions, because they carry the lesson. `verb_of` and `arg_of` split on the *first* `|` only, so a message containing `|` (for example `SAY|a|b`) keeps its tail intact. Splitting on every `|` with `split` would have truncated it. The `name_of` lookup uses delimiter-wrapped keys (`;1=ada;`) so that client `1` cannot be confused with client `11`.

The newline problem

A frame ends at a newline, and `net_send` does not escape one. If a client puts a newline inside its text, the host receives *two* frames. We proved it:

GUI

```
// ling run nl.ling --role host   /   ling run nl.ling
fn host() {
  net_host(18137)
  bind quiet = 0
  while quiet < 30 {
    bind m = net_recv()
    if m == "" { sleep(50) bind quiet = quiet + 1 } else { println("host got [", m,
"]") bind quiet = 0 }
  }
  net_close()
}

fn client() {
  net_join("127.0.0.1", 18137)
  bind n = 0
  while net_status() != 2 && n < 100 { sleep(50) bind n = n + 1 }
  net_send("SAY|first line\nSAY|forged second line")
  net_send("SAY|" + text_hex("safe\nbinary\n"))
  sleep(500)
  net_close()
}

bind start = do {
  if cli_arg("role", "client") == "host" { host() } else { client() }
}
```

OUTPUT

```
host got [SAY|first line]
host got [SAY|forged second line]
host got [SAY|736166650a62696e6172790a]
```

The first `net_send` contained `SAY|first line\nSAY|forged second line` and produced two valid-looking `SAY` frames, so a client could inject a frame the user never typed. The second send hex-encodes its payload with `text_hex`, which turned `safe\nbinary\n` into `736166650a62696e6172790a` and arrived as a single frame; the receiver reverses it with `hex_text`, which returns `ok(text)` or `bad(message)`. Choose one of these defences on day one: reject any payload containing a newline before sending, or encode every free-form payload (hex or base64), and *always* re-check on the receiving side, since a hostile client does not run your sending code.

> TIP
>
> Write down the protocol as a table before you write code: verb, direction, argument, reply, error cases, size limit. If you cannot fill every cell, the protocol is not ready. A table of eight lines is also the best documentation a peer implementer in another language will ever get.

Remember also the limits that the runtime imposes. The host accepts up to 64 clients, each inbound queue holds 4,096 lines and drops the oldest when full, and nothing is encrypted. A slow consumer therefore loses messages rather than blocking the producer, which is the right trade-off for a game and the wrong one for a ledger. Add sequence numbers if loss matters.

`lingtp`: pinned keys and a deliberately tiny protocol

`lingtp` aims to do what HTTPS does, without certificate authorities. The server owns an ML-DSA-87 signing identity in a key file. On the first connection a client records the server's public key in `.lingtp/known_hosts` (relative to the current directory). From then on it accepts only that key. This is trust on first use, the SSH model, and its security is exactly as good as your first connection.

The application layer is minimal on purpose: a request is one string (the path, query included), a response is one string. A POST sends `path + "\n" + body`, so a handler distinguishes the two by looking for a newline. The program below is a key-value store that speaks that convention. Note how it splits the combined string itself, and how it refuses keys containing `.`, `/` or a backslash: the key becomes part of a file name, and without that check a client could write outside the directory.

GUI

```
// A tiny key-value store behind lingtp. Paths:  /get?k=NAME   POST /put?k=NAME (body =
value)
fn handle(path) {
  bind nl = str_find(path, "\n")
  bind head = path
  bind body = ""
  if nl >= 0 {
    bind head = substr(path, 0, nl)
    bind body = substr(path, nl + 1, len(path))
  }
  bind q = ""
  bind qat = str_find(head, "?")
  if qat >= 0 { bind q = substr(head, qat + 1, len(head)) }
  bind route = head
  if qat >= 0 { bind route = substr(head, 0, qat) }
  bind k = query_param(q, "k", "")
  if k == "" { return "ERR missing k" }
  if str_find(k, ".") >= 0 || str_find(k, "/") >= 0 || str_find(k, "\\") >= 0 {
    return "ERR bad key"
  }
  if route == "/put" && nl >= 0 {
    write_file("kv_" + k + ".txt", body)
    return "stored " + k
  }
  if route == "/get" {
    if !file_exists("kv_" + k + ".txt") { return "ERR no such key" }
    return read_file("kv_" + k + ".txt")
  }
  return "ERR unknown route"
}

bind start = do {
  lingtp_config(18131, cli_arg("key", "host.key"))
  lingtp_route("/", handle)
  println("serving")
  lingtp_serve()
}
```

```
GUI
bind start = do {
  bind port = 18131
  println(lingtp_post("127.0.0.1", port, "/put?k=motd", "hello over post-quantum TCP"))
  println("status ", lingtp_get_status())
  println(lingtp_get("127.0.0.1", port, "/get?k=motd"))
  println(lingtp_get("127.0.0.1", port, "/get?k=nope"))
  println(lingtp_get("127.0.0.1", port, "/zzz?k=a"))
  println(lingtp_get("127.0.0.1", port, "/get?k=../secret"))
  println("status ", lingtp_get_status())
}
```

With the server running (`ling run srv.ling`), the client prints:

```
OUTPUT
stored motd
status 1
hello over post-quantum TCP
ERR no such key
ERR unknown route
ERR bad key
status 1
```

`lingtp_get_status()` is the thing to check. It is `1` for success, `0` for a connect or handshake failure, `-1` for an invalid server signature and `-2` when the host key changed. Application-level `ERR` strings above are *successful transport* with a failed request: your protocol must carry its own error vocabulary, because there are no status codes on this wire.

What a pinned key buys you

Stop the server, delete `host.key` so it generates a fresh identity, and start it again. Run the same client:

```
OUTPUT
$ ling run cli.ling

status -2




status -2
```

The blank lines are the empty response bodies (one per request), and `-2` is "host key changed, connection refused". The client never sent a byte of application data to the impostor. An attacker who can sit between you and the server cannot silently replace its key after your first contact. If you really did rebuild the server, delete the matching line in `.lingtp/known_hosts` on each client.

> **WARNING**
>
> The host key file is a plain hex seed. Whoever reads it can impersonate your server. Protect it like an SSH private key, back it up (losing it changes the identity and every client refuses you), and never put it in a repository.

Two implementations that cannot talk

Here is the trap that costs people an afternoon. The Ling runtime implements `lingtp` itself, with a bare path string and a hex-seed key file. The `ling-net` crate has a separate `lingtp` module with a JSON request envelope and a raw 32-byte key file. They share cryptography but not wire format. We started the Rust server (next section) and pointed a Ling `lingtp_get` at it:

```
GUI
bind start = do {
  bind body = lingtp_get("127.0.0.1", 18132, "/health")
  println("body: [", body, "]  status: ", lingtp_get_status())
}
```

```
OUTPUT
body: []  status: 0
```

Status `0`, empty body: no message tells you *why*, because to the client a handshake with a foreign implementation is just a failed handshake. The rule is simple and absolute: **use one implementation on both ends**. Ling client with Ling server, or Rust client with Rust server.

SSH: a protocol endpoint, not a shell

The `ssh_*` builtins wrap a real SSH-2 implementation. The *client* can talk to any OpenSSH server; the *server* accepts exactly one configured account by password, persists an Ed25519 host key, and queues what clients send for your code to answer. It does not run commands; your program is the command interpreter. Over loopback with the Ling client and Ling server, the parts we could confirm were the connection and authentication states:

```
GUI
bind start = do {
  println("wrong password: ", ssh_connect("127.0.0.1", 18135, "deploy", "nope"))
  println("right password: ", ssh_connect("127.0.0.1", 18135, "deploy", "correct
horse"))
  println("client status: ", ssh_client_status())
  ssh_close()
  println("closed: ", ssh_client_status())
  println("nobody home: ", ssh_connect("127.0.0.1", 18136, "deploy", "x"))
}
```

```
OUTPUT
wrong password: 0
right password: 1
client status: 2
closed: 0
nobody home: -1
```

The server side logged `authfail:deploy`, then a disconnect for each client. The return codes are `1` authenticated, `0` bad credentials, `-1` no connection or handshake failure and `-2` a changed host key. What we could *not* get working is command execution between the two Ling halves: `ssh_exec` returned an empty string and exit code `-1` against `ssh_serve`, although the server received the command. We could not resolve this, so we do not recommend the pairing. If you need remote commands, use the Ling client against a real OpenSSH server, which is the documented use, and treat that path as untested by us.

Only password authentication exists on both sides: there is no key-based login. Read passwords from the environment with `env_get`, never from the command line.

`p2p`: nodes with identities

The `p2p_*` family exposes LWP-1, the transport from the Ling wallet project, implemented in `ling_net::lwp`. There is no client and no server: every node is the same program. A node has a long-term ML-DSA-87 identity, an optional listening address and a list of peers to dial. Peers authenticate each other during a hybrid X25519 plus ML-KEM-1024 handshake, and every frame is padded to a fixed bucket size and sealed, so an observer cannot read content or infer message sizes precisely. Payloads are opaque: *you* decide what a message means. The transport reserves message kinds below 16 for itself, and gives you 16 to 255.

Two mechanics matter before any code. Everything is **polling**: `p2p_poll` returns a list of events, and you must keep calling it, because a full queue applies back-pressure to every peer. And **payloads are hex strings**, which is why `text_hex` and `hex_text` appear in every example. We confirmed the event and return shapes on a two-node loopback network:

```
GUI
fn unwrap(r) { return match r { ok(v) => v, bad(e) => "ERR:" + e } }

bind start = do {
  bind a = unwrap(p2p_start([["magic_hex", "4c494e47"], ["network", 1], ["listen",
"127.0.0.1:0"], ["data_dir", "n_a"], ["fast_timers", true]], ""))
  bind b = unwrap(p2p_start([["magic_hex", "4c494e47"], ["network", 1], ["listen", ""],
["data_dir", "n_b"], ["peers", [p2p_local_addr(a)]], ["fast_timers", true]], ""))
  bind evs = p2p_poll(a, 8, 3000)
  println("events on A: ", len(evs))
  println("first event kind: ", evs[0][0], ", inbound: ", evs[0][3])
  println("fingerprint shape: ", len(evs[0][4]), " chars")
  bind ps = p2p_peers(b)
  bind n = 0
  while len(ps) == 0 && n < 50 {
    sleep(100)
    bind ps = p2p_peers(b)
    bind n = n + 1
  }
  println("B peers: ", len(ps), ", inbound flag: ", ps[0][2], ", seed: ", ps[0][4])
  println("netgroup: ", p2p_netgroup(b, ps[0][0]))
  println("bad kind: ", p2p_send(b, ps[0][0], 3, "00"))
  println("bad peer: ", p2p_send(b, 99, 16, "00"))
  println("config typo: ", unwrap(p2p_start([["magic_hex", "4c494e47"], ["network", 1],
["data_dir", "n_c"], ["listn", ""]], "")))
  println("missing magic: ", unwrap(p2p_start([["network", 1], ["data_dir", "n_c"]],
"")))
  p2p_shutdown(a)
  println("shutdown twice: ", p2p_shutdown(b), " ", p2p_shutdown(b))
}
```

```
OUTPUT
events on A: 1
first event kind: connected, inbound: true
fingerprint shape: 19 chars
B peers: 1, inbound flag: false, seed: false
netgroup: local:127.0.0.1:2980
bad kind: false
bad peer: false
config typo: ERR:unknown config key: listn
missing magic: ERR:config: magic_hex is required
shutdown twice: true false
```

(The port inside `netgroup` varies per run.) Notice the error behaviour, which is stricter than the rest of Ling: unknown config keys and missing required keys return `Err(message)`, not a silent `()`; sending a reserved kind or to an unknown peer returns `false`. After `p2p_shutdown`, a second shutdown returns `false`. `p2p_start` requires the `"p2p"` capability, which the main thread has.

The configuration is a list of `[key, value]` pairs. The required keys are `magic_hex` (four bytes in hex, which separates your network from every other: peers with different magic never connect), `network` (0 to 255) and `data_dir`. `listen` is an `"ip:port"` string, with port `0` meaning "pick one" (read it back with `p2p_local_addr`), or `""` for an outbound-only node. `peers` is a list of addresses to dial. `fast_timers` shortens internal timers and is intended for tests; production nodes should omit it.

A gossip network

The classic pattern is gossip: a node that receives a new message relays it to its other peers, and drops anything it has already seen. Our network has three nodes. Node A listens. B and C know only A. B broadcasts one message, A relays it, and C receives it without ever being connected to B.

GUI

```
// Three nodes on loopback: B and C only know A. A relays what it hears,
// so a message sent by B reaches C without B and C ever connecting.
fn unwrap(r) { return match r { ok(v) => v, bad(e) => "ERR:" + e } }

fn start_node(dir, listen, peers) {
  return unwrap(p2p_start([
    ["magic_hex", "4c494e47"], ["network", 1],
    ["listen", listen], ["data_dir", dir],
    ["peers", peers], ["fast_timers", true]
  ], ""))
}

fn wait_peers(net, want) {
  bind n = 0
  while len(p2p_peers(net)) < want && n < 100 {
    sleep(100)
    bind n = n + 1
  }
  return len(p2p_peers(net))
}

// Drain one node. Relays unseen kind-20 messages to everyone except the sender.
// seen is a text of "|id|id|" because lists cannot be updated across calls.
fn pump(name, net, seen, relay) {
  bind evs = p2p_poll(net, 16, 300)
  bind i = 0
  while i < len(evs) {
    bind ev = evs[i]
    if ev[0] == "message" && ev[2] == 20 {
      bind id = hash_hex(ev[3])
      if str_find(seen, id) < 0 {
        bind seen = seen + id + "|"
        println(name, " got: ", unwrap(hex_text(ev[3])), " (from peer ", ev[1], ")")
        if relay { p2p_broadcast(net, 20, ev[3], ev[1]) }
      } else {
        println(name, " dropped duplicate")
      }
```

```
    }
    bind i = i + 1
  }
  return seen
}

bind start = do {
  bind a = start_node("node_a", "127.0.0.1:0", [])
  bind addr = p2p_local_addr(a)
  bind b = start_node("node_b", "", [addr])
  bind c = start_node("node_c", "", [addr])
  println("A peers: ", wait_peers(a, 2), "  B peers: ", wait_peers(b, 1), "  C peers:
", wait_peers(c, 1))

  p2p_broadcast(b, 20, text_hex("hello from B"), -1)
  bind sa = "|"
  bind sb = "|"
  bind sc = "|"
  bind round = 0
  while round < 4 {
    bind sa = pump("A", a, sa, true)
    bind sc = pump("C", c, sc, false)
    bind sb = pump("B", b, sb, false)
    bind round = round + 1
  }
  p2p_shutdown(a)
  p2p_shutdown(b)
  p2p_shutdown(c)
  println("done")
}
```

```
OUTPUT
A peers: 2  B peers: 1  C peers: 1
A got: hello from B (from peer 1)
C got: hello from B (from peer 1)
done
```

The `except_peer` argument to `p2p_broadcast` is the sender's id: A relays to everyone but the peer it heard from, so B does not receive its own message back. The duplicate guard matters because in a mesh with cycles, relays loop forever without it. Here `seen` is a delimited string of message hashes (`hash_hex`), carried in and out of `pump`, because a Ling function cannot modify a list it was passed. The output is the same under `--interp`. Run the whole program a few times: node ids are per-node, so "from peer 1" means *A's id as seen by C*, not a global name.

> NOTE
>
> A pitfall we hit while writing this: the first draft tested membership with `contains(seen, id)`. On the default JIT that call quietly returned something truthy and every message was dropped as a duplicate ("A dropped duplicate"), and on `--interp` it was an unknown function. `str_find(seen, id) < 0` behaves identically on both. When something works on one backend and not the other, check the builtin exists before suspecting your logic.

What we did not test

`p2p_relay` and `p2p_mark_fluffed` (Dandelion++ transaction relay), `p2p_misbehaving` banning, SOCKS5 proxying (for Tor), signed seed lists (`p2p_seeds_sign` and `p2p_seeds_verify`), and NAT behaviour were not exercised. There is no NAT traversal in the transport at all: a node behind a router must be reachable by port forwarding. `p2p_fingerprint` exists in the repository but evaluates to `()` in the installed 2030.1.53 binary, so rebuild from source to use it. Peer discovery beyond manual `peers` and seed lists is up to you.

`ling-net` from Rust

The `ling-net` crate has opt-in features: `http` (default), `websocket`, `quic`, `grpc`, `lingtp` and `lwp`. The last two are the ones that matter here, because `lwp` is *the same code* that backs `p2p_*`. That makes a strong claim testable: a Rust node should be able to join a network of Ling nodes. Here is a Rust program with three modes. `roundtrip` runs a `lingtp` server and client in one process. `serve` runs a `lingtp` server alone (the impostor from the previous section). `lwp-dial` joins a Ling `p2p` node, sends one message and waits for a reply.

```
[dependencies]
ling-net   = { path = "C:/Users/User/Programs/ling/crates/ling-net", default-features =
false, features = ["lingtp", "lwp"] }
serde_json = "1"

[workspace]
```

```rust
use ling_net::lingtp::{client, server, LingtpRequest, LingtpResponse};
use ling_net::lwp::crypto::identity::IdentityKey;
use ling_net::lwp::{NetConfig, NetEvent, Network};
use std::net::TcpListener;
use std::time::{Duration, Instant};

fn handler(req: &LingtpRequest) -> LingtpResponse {
    match (req.method.as_str(), req.path.as_str()) {
        ("GET", "/health") => LingtpResponse::text(200, "ok"),
        ("POST", "/echo") => match req.json::<serde_json::Value>() {
            Ok(v) => LingtpResponse::json(200, &serde_json::json!({ "you_sent": v })),
            Err(e) => LingtpResponse::error(400, format!("bad json: {e}")),
        },
        _ => LingtpResponse::error(404, "not found"),
    }
}

/// Rust server and Rust client in one process, on an ephemeral port.
fn roundtrip() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:0")?;
    let port = listener.local_addr()?.port();
    let identity = server::load_or_generate_identity("rt_host_key")?;
    std::thread::spawn(move || server::serve_on(listener, identity, handler));

    let mut conn = client::connect("127.0.0.1", port)?; // one handshake, many requests
    let r = conn.request(&LingtpRequest::get("/health"))?;
    println!("GET /health -> {} {}", r.status, r.body);
    let post = LingtpRequest::post("/echo", "").with_json_body(&serde_json::json!({"n":
7}))?;
    let r = conn.request(&post)?;
    println!("POST /echo -> {} {}", r.status, r.body);
    let r = conn.request(&LingtpRequest::get("/nope"))?;
    println!("GET /nope -> {} {}", r.status, r.body);
    Ok(())
}

/// Dial a Ling `p2p_*` node, send one application message, wait for the reply.
fn lwp_dial(addr: &str) -> Result<(), Box<dyn std::error::Error>> {
```

```
    let cfg = NetConfig {
        network_magic: *b"LING",
        network: 1,
        listen: None,
        data_dir: "rust_node".into(),
        seeds: vec![],
        socks5_proxy: None,
        max_outbound: 8,
        max_inbound: 8,
        identity: IdentityKey::generate(),
        ephemeral_outbound_identity: true,
        manual_peers: vec![addr.to_string()],
    };
    let (net, rx) = Network::start(cfg)?;
    let deadline = Instant::now() + Duration::from_secs(40);
    let mut sent = false;
    while Instant::now() < deadline {
        match rx.recv_timeout(Duration::from_millis(250)) {
            Ok(NetEvent::Connected(p)) => {
                println!("connected to {} (inbound={}) fp {}", p.addr, p.inbound,
p.identity_fingerprint);
                net.send(p.id, 21, b"hello from Rust".to_vec());
                sent = true;
            }
            Ok(NetEvent::Message { kind, payload, .. }) => {
                println!("message kind {kind}: {}", String::from_utf8_lossy(&payload));
                break;
            }
            Ok(_) | Err(_) => {}
        }
    }
    if !sent {
        println!("never connected");
    }
    net.shutdown();
    Ok(())
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
```

```
    let args: Vec<String> = std::env::args().collect();
    match args.get(1).map(String::as_str) {
        Some("roundtrip") => roundtrip(),
        Some("serve") => {
            let port: u16 = args[2].parse()?;
            let id = server::load_or_generate_identity("rust_host_key")?;
            server::serve("127.0.0.1", port, id, handler)?;
            Ok(())
        }
        Some("lwp-dial") => lwp_dial(&args[2]),
        _ => {
            eprintln!("usage: netrs roundtrip | serve PORT | lwp-dial ADDR");
            Ok(())
        }
    }
}
```

The Rust `lingtp` server is a closure from request to response, and the request is a structured value with `method`, `path`, `headers` and `body`, which is the envelope that Ling's runtime does not speak. Running `roundtrip`:

```
OUTPUT
GET /health -> 200 ok
POST /echo -> 200 {"you_sent":{"n":7}}
GET /nope -> 404 {"error":"not found"}
```

Notice that this wire does carry status codes and JSON, which is an advantage of the Rust implementation, and a reason to choose it if both ends are Rust. A `Connection` performs one handshake and then serves many requests (keep-alive).

Rust meets Ling on `p2p`

Now the interoperability test. The Ling side is a node listening on a fixed port that answers any kind-21 message with a kind-22 reply:

GUI

```
// A Ling p2p node that acknowledges kind-21 messages with a kind-22 reply.
fn unwrap(r) { return match r { ok(v) => v, bad(e) => "ERR:" + e } }

bind start = do {
  bind net = unwrap(p2p_start([["magic_hex", "4c494e47"], ["network", 1], ["listen",
"127.0.0.1:18134"], ["data_dir", "lnode"]], ""))
  println("ling node listening on ", p2p_local_addr(net))
  bind n = 0
  bind done = false
  while !done && n < 90 {
    bind evs = p2p_poll(net, 16, 500)
    bind i = 0
    while i < len(evs) {
      bind ev = evs[i]
      if ev[0] == "connected" { println("peer ", ev[1], " connected, inbound=", ev[3])
}
      if ev[0] == "message" && ev[2] == 21 {
        println("kind 21: ", unwrap(hex_text(ev[3])))
        p2p_send(net, ev[1], 22, text_hex("ack: " + unwrap(hex_text(ev[3]))))
        bind done = true
      }
      bind i = i + 1
    }
    bind n = n + 1
  }
  sleep(1500)
  p2p_shutdown(net)
}
```

Start it, then run `netrs lwp-dial 127.0.0.1:18134`. The Rust node dials, authenticates, sends `hello from Rust` as kind 21 and prints the reply. Both sides' logs:

OUTPUT

```
connected to 127.0.0.1:18134 (inbound=false) fp 3b89-e4b3-7e48-d634
message kind 22: ack: hello from Rust
```

```
OUTPUT
ling node listening on 127.0.0.1:18134
peer 1 connected, inbound=true
kind 21: hello from Rust
```

It connected and completed the handshake in under a second. This is the strongest result in the chapter: unlike `lingtp`, the `lwp` code is shared, so a Rust node and a Ling node are the same protocol. The fingerprint Rust printed belongs to the *remote* peer, the Ling node, whose identity was a fresh random one because we gave it no `identity_seed_hex`; a node that wants a stable, recognisable identity must be given a seed. Likewise the Rust node dialled with `ephemeral_outbound_identity: true`, a new random identity per outbound connection, which is the privacy default. Rust's `Network::start` returns a handle and a `crossbeam` receiver of `NetEvent` values (`Connected`, `Disconnected`, `Message`, `StemTx`), the same four events that Ling's `p2p_poll` flattens into lists.

> **TRY IT**
>
> Change the `magic_hex` on the Ling node to `4c494e48` and rerun the Rust dialer. It prints `never connected` after its 40 second deadline (we ran it), because peers with different network magic do not talk, and the Ling node logs nothing. Then restore it. This is the cheapest test of your network separation.

Verified and not verified

This table is the honest summary of this chapter and of our testing. "Run" means we executed it and observed the stated result on loopback.

Feature	Status
`net_*` host and clients, multi-client chat, `net_send_to`	Run; JIT and `--interp` identical
Embedded newline injection in `net_send`	Run; reproduced
`lingtp_*` get, post, key pinning, changed-key refusal (`-2`)	Run
Ling `lingtp` against Rust `lingtp` server	Run; fails with status `0`
`ling-net` `lingtp` client and server round trip	Run
`ssh_connect` statuses, wrong password, server events	Run
`ssh_exec` against `ssh_serve`	Run; failed (empty result, exit `-1`); unresolved
`p2p_*` two- and three-node loopback, gossip with relay	Run

Feature	Status
Rust `lwp` node joining a Ling `p2p` node	Run
`p2p_relay`, `p2p_mark_fluffed`, bans, SOCKS5, seed lists	Not run
`net_announce`, `net_discover`, `net_test`	Not run
Anything across a real network, NAT, or with packet loss	Not run
WebSocket, QUIC and gRPC features of `ling-net`	Not run

Chapter summary

- Pick the transport by who controls the peers: HTTP with TLS for the open world, `lingtp` for private service-to-service calls with pinned keys, `ssh_*` for an administrative channel, `p2p_*` for equal nodes, `net_*` only for trusted LANs.
- A transport gives you bytes; you design the protocol. Write a table of verbs, replies, errors and limits. Split on the first delimiter only, and defend against embedded newlines by rejecting or encoding payloads.
- `lingtp` pins the server key on first use. Status `-2` means the key changed and the connection was refused. Its protocol has no status codes, so carry your own.
- The Ling and Rust `lingtp` implementations are incompatible; use the same one at both ends. The Ling and Rust `p2p` implementations are the same code and interoperate.
- `p2p_*` is polled, payloads are hex, message kinds start at 16, and configuration errors return `Err` rather than failing silently. Gossip needs an explicit seen-set and the `except_peer` argument.
- Loopback success is evidence that the code works, not that your network will. Test across real machines, and keep the verified-versus-not list for your own deployment.

Exercises

1. **Harden the chat.** Make the host reject any `SAY` payload longer than 200 characters (it already does) and any frame containing a character outside printable ASCII, replying `ERR|bad character`. Then change the client to hex-encode its text and the host to decode with `hex_text`. What does the host do with a frame that is not valid hex?
2. **Count the hops.** Extend the gossip program so each message carries a hop count: payload `hops|text`. A node increments the count when relaying and refuses to relay beyond three hops. Which function lets you split the payload safely, and why must you not use `split(text, "|")`?

3. **Detect an impostor.** Write a `lingtp` client that calls the server twice, with a server restart (and key regeneration) between the calls, and prints either `trusted` or `KEY CHANGED` based on `lingtp_get_status()`. What must the client do to recover legitimately, and what must it check first?

Answers

1. Add a check before the verb dispatch: loop over the characters of the argument with an index and `substr(arg, i, 1)`, and compare against a printable range, rejecting with `net_send_to(id, "ERR|bad character")`. Because `substr` and comparison on single characters can be awkward, a simpler approach is to require the payload be hex: the client sends `SAY|` plus `text_hex(text)`, and the host calls `hex_text(payload)`. `hex_text` returns `ok(text)` for valid hex and `bad("argument 1 is not valid hex")` for anything else, so the host matches on it and answers `ERR|bad encoding` on the error arm. Hex encoding also makes newline injection impossible, because the encoded form contains only `0-9a-f`.
2. Use the same `str_find` and `substr` pair as `verb_of` and `arg_of` in the chat program: find the first `|`, take `to_number` of the prefix as the hop count and the remainder as the text. `split(text, "|")` would break the text apart if it contains `|`, silently losing everything after the second one. In `pump`, replace the relay with `if hops < 3 { p2p_broadcast(net, 20, text_hex((hops + 1) + "|" + text), ev[1]) }`. Because the seen-set is keyed on the hash of the *whole payload*, the incremented message has a new hash, so remember to key the seen-set on the text alone, or a node may re-accept the same message with a different hop count.
3. Call `lingtp_get` before the restart and print `trusted` when `lingtp_get_status()` is `1`. After the server regenerates its key, call it again: the status is `-2`, so print `KEY CHANGED`. To recover legitimately, first verify out of band that the server really was rebuilt (ask the operator, compare fingerprints through a second trusted channel), and only then delete that server's line from `.lingtp/known_hosts`. Never automate the deletion: that turns key pinning into a prompt that any attacker can click through.

CHAPTER 7

AI and Tensors

What you'll learn

"AI" in Ling means four different things with four different costs, and the skill is picking the smallest one that solves your problem. This chapter walks up the ladder. You will train a neural network from Ling code and save it, write behaviour trees that drive a non-player character, teach a miniature language model a character's voice, and combine all three into a working sentry. Then you will see how local large-language-model and vision inference plug in, what they need that the default build does not have, and how to design your program so it degrades gracefully without them.

The second half drops into Rust. The same engines are the `ling-ai` crate, which adds a small tensor type and lets you write your own training loop. We finish with a network trained in Rust and loaded by a Ling script, byte for byte.

Everything in the first half runs headless: no window, no model files, no network. That is deliberate, and it is the habit to copy. AI logic that can only be tested with a graphics window open is logic you will not test.

Four layers, one handle convention

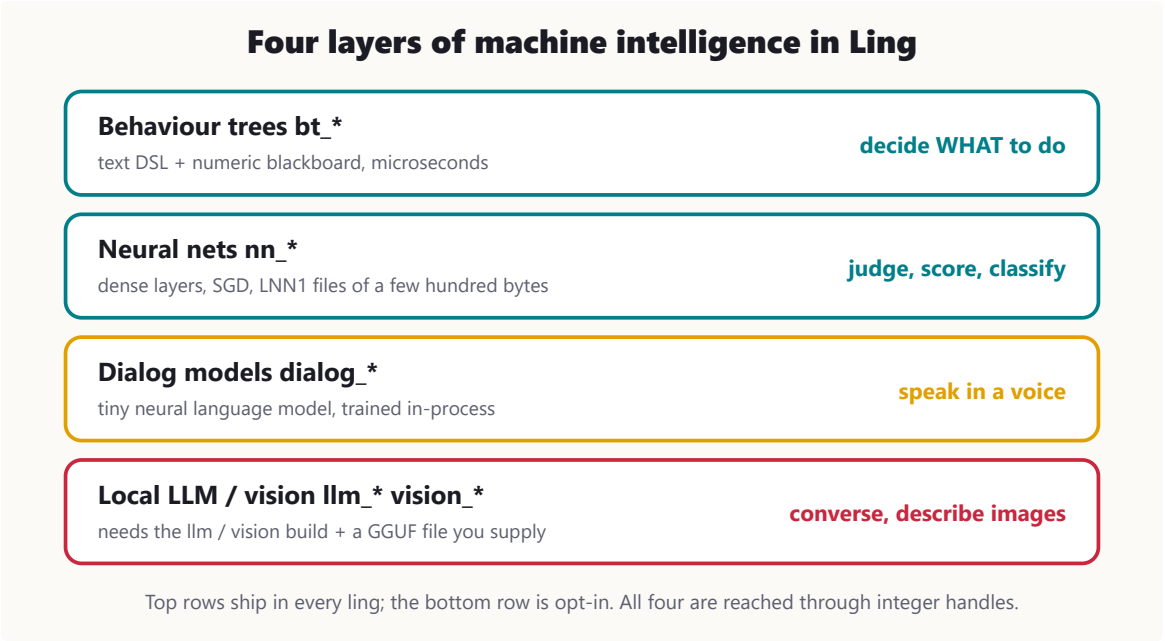


Figure: the AI layers, cheapest at the top. The top three ship in every `ling`; the bottom needs a special build and a model file.

Behaviour trees (`bt_*`), neural networks (`nn_*`) and dialog models (`dialog_*`) are pure Rust, CPU only, with no model files; their state is kilobytes. Local language-model chat (`llm_*`) and image understanding (`vision_*`) load a multi-hundred-megabyte checkpoint and exist only in a `ling` built with the Cargo features `llm` and `vision`.

All of them follow one convention: creating something returns a small integer handle, and a bad handle is never fatal. A failed load returns `-1`; every other call on a bad handle returns a neutral value (`[]`, `0`, `""`, `false`). Handles are never reused, so a program that creates thousands of networks in a loop leaks them until exit. Create things once, at start-up.

> **WARNING**
>
> In the default `ling run` (JIT) a builtin that is not compiled in evaluates silently to `()`, exactly like a misspelt name. Under `--interp` the same call is a hard error, `unknown function 'llm_load'`. Both are useful, but only the first lets you probe for a feature at run time. We will use that later.

Neural networks with nn_*

The network is a plain feed-forward stack: you say how many numbers go in, append dense layers, then train one example at a time with stochastic gradient descent on a squared-error loss. There is no autograd graph and the inference path allocates nothing, so a forward pass is cheap enough to run every frame. Weights start from the seed you pass to `nn_new`, so a run is reproducible.

Activation names are `"relu"`, `"tanh"`, `"sigmoid"`, and anything else means linear. A classic first task is XOR, which a single layer cannot learn but one hidden layer can:

```
// Train XOR and watch the loss fall.
bind start = do {
  bind net = nn_new(2, 7)
  nn_dense(net, 6, "tanh")
  nn_dense(net, 1, "sigmoid")
  bind epoch = 0
  while epoch <= 1500 {
    bind total = nn_train(net, [0, 0], [0], 0.5) + nn_train(net, [0, 1], [1], 0.5) +
nn_train(net, [1, 0], [1], 0.5) + nn_train(net, [1, 1], [0], 0.5)
    if epoch == 0 || epoch == 100 || epoch == 500 || epoch == 1500 {
      println("epoch ", epoch, " loss ", round(total * 10000) / 10000)
    }
    bind epoch = epoch + 1
  }
  println("0 xor 1 -> ", round(nn_forward(net, [0, 1])[0] * 100) / 100)
  println("1 xor 1 -> ", round(nn_forward(net, [1, 1])[0] * 100) / 100)
}
```

```
OUTPUT
epoch 0 loss 0.5818
epoch 100 loss 0.1081
epoch 500 loss 0.0037
epoch 1500 loss 0.0009
0 xor 1 -> 0.98
1 xor 1 -> 0.03
```

`nn_train` returns the loss *before* the update, `0.5 * sum((output - target)^2)`, so summing the four returns gives the epoch's loss. Watch the shape: a steep fall in the first hundred epochs, then a long tail. Print this number while you tune; a loss that stalls high means the learning rate is wrong, the layer is too small, or the data is unlearnable.

Real tasks need more than four rows. The next program classifies random points as inside or outside a circle, which no linear model can do. It generates its own data with a tiny linear congruential generator, because `rand()` cannot be seeded and a test should be repeatable. It trains on 30,000 points, then measures accuracy on 1,000 *fresh* ones, which is the only number that means anything:

```
// Classify points: inside the circle of radius 0.6 around the origin?
fn lcg(s) { return (s * 75 + 74) % 65537 }

fn sample(s) { return s / 65537 * 2 - 1 }

fn inside(x, y) {
  if x * x + y * y < 0.36 { return 1 }
  return 0
}

bind start = do {
  bind net = nn_new(2, 3)
  nn_dense(net, 12, "tanh")
  nn_dense(net, 8, "tanh")
  nn_dense(net, 1, "sigmoid")
  bind s = 1
  bind step = 0
  bind last = 0
  while step < 30000 {
    bind s = lcg(s)
    bind x = sample(s)
    bind s = lcg(s)
    bind y = sample(s)
    bind last = nn_train(net, [x, y], [inside(x, y)], 0.1)
    bind step = step + 1
  }
  // evaluate on fresh points
  bind right = 0
  bind n = 0
  while n < 1000 {
    bind s = lcg(s)
    bind x = sample(s)
    bind s = lcg(s)
    bind y = sample(s)
    bind guess = 0
    if nn_forward(net, [x, y])[0] > 0.5 { bind guess = 1 }
    if guess == inside(x, y) { bind right = right + 1 }
    bind n = n + 1
```

```
  }
  println("held-out accuracy: ", right / 10, "%")
  println("centre:  ", round(nn_forward(net, [0, 0])[0] * 100) / 100)
  println("corner:  ", round(nn_forward(net, [0.9, 0.9])[0] * 100) / 100)
  println("saved: ", nn_save(net, "circle.lnn"))
  bind again = nn_load("circle.lnn")
  println("reloaded matches: ", nn_forward(again, [0.1, 0.2])[0] == nn_forward(net,
[0.1, 0.2])[0])
  println("file bytes: ", file_size("circle.lnn"))
  remove_file("circle.lnn")
}
```

```
OUTPUT
held-out accuracy: 95.8%
centre:  1
corner:  0
saved: true
reloaded matches: true
file bytes: 635
```

Two habits are visible here. Inputs are scaled to roughly -1..1; networks train badly on raw pixel counts or world coordinates in the thousands. And evaluation uses points the network never saw. Training loss alone will happily report success for a network that has merely memorised.

`nn_save` writes the compact `LNN1` binary format (little-endian, 635 bytes for this net) and `nn_load` returns a new handle, or `-1` for a missing or invalid file.

> **WARNING**
>
> An `.lnn` file has no checksum or signature, and the loader does not authenticate it. Never load one from an untrusted source inside a service. If you distribute models, sign them with the techniques from *Cryptography in Practice* and verify before loading.

> **TIP**
>
> `nn_forward` pads or truncates its input to the network's input width without complaint. A mismatch between training and inference code therefore produces wrong answers, not errors. Wrap the feature vector in one function that both sides call.

Behaviour trees with bt_*

A neural network answers "how much?". A behaviour tree answers "what do I do?". You write the tree in a small text language and tick it against a blackboard of numeric facts. A tick returns the name of the action chosen; your program stays in charge of performing it.

The grammar has `selector` (first child that succeeds), `sequence` (all children must succeed), `parallel` (tick all), `not <child>`, conditions `?key OP value`, and actions `!name`. Here is a guard:

```
// A guard that fights, flees, reloads or patrols, depending on the blackboard.
bind start = do {
  bind brain = bt_build("
    selector {
      sequence { ?health < 0.25 ?enemy_near > 0 !flee }
      sequence { ?enemy_near > 0 ?ammo >= 1 !shoot }
      sequence { ?enemy_near > 0 not ?ammo >= 1 !reload }
      sequence { ?alert > 0 !investigate }
      !patrol
    }
  ")
  bt_set(brain, "health", 1)
  println("quiet:      ", bt_tick(brain))
  bt_set(brain, "alert", 1)
  println("noise:      ", bt_tick(brain))
  bt_set(brain, "enemy_near", 1)
  bt_set(brain, "ammo", 3)
  println("enemy+ammo: ", bt_tick(brain), " status ", bt_status(brain))
  bt_set(brain, "ammo", 0)
  println("no ammo:    ", bt_tick(brain))
  bt_set(brain, "health", 0.2)
  println("hurt:       ", bt_tick(brain))
  bt_set(brain, "enemy_near", 0)
  println("safe again: ", bt_tick(brain))
  println("bad spec:   ", bt_build("selector { !a } extra"))
  println("bad handle: [", bt_tick(99), "] status ", bt_status(99))
}
```

```
OUTPUT
quiet:      patrol
noise:      investigate
enemy+ammo: shoot status 1
no ammo:    reload
hurt:       flee
safe again: investigate
bad spec:   -1
bad handle: [] status 0
```

Read the tree top to bottom as priorities. Flee beats shoot, shoot beats reload, and so on down to patrol, the fallback. Each `sequence` is "if all these conditions hold, do this action". Notice `not ?ammo >= 1`: `not` takes exactly one child and no braces. Writing `not { ?ammo >= 1 }` fails to parse, and `bt_build` answers `-1`, as the bad-spec line shows (trailing garbage after the tree also fails).

The first line of the program sets `health` to 1, and that is not decoration. A key that was never set reads as 0, so an unset `health` satisfies `?health < 0.25`, and a guard that has simply not been told its health yet runs away from everything. Initialise every fact the tree reads before the first tick. The next program shows the trap, and one more oddity of `parallel`:

```
bind start = do {
  // an unset fact reads as 0, so "health < 0.25" is true before anyone sets health
  bind t = bt_build("selector { sequence { ?health < 0.25 !flee } !patrol }")
  println(bt_tick(t))
  bt_set(t, "health", 1)
  println(bt_tick(t))
  // == uses a tiny epsilon; parallel ticks every child
  bind p = bt_build("parallel { ?a == 1 ?b == 2 !both }")
  bt_set(p, "a", 1)
  println("a only: [", bt_tick(p), "] ", bt_status(p))
  bt_set(p, "b", 2)
  println("a and b: ", bt_tick(p), " ", bt_status(p))
}
```

```
OUTPUT
flee
patrol
a only: [both] 0
a and b: both 1
```

With only `a` set, the `parallel` node fails (status 0) but the tick still reports `both`, because every child is ticked and the action child records its name. When you use `parallel`, check `bt_status` as well as the action string. For ordinary decision logic, stick to `selector` and `sequence`, where a failed branch never reports its action.

> **NOTE**
>
> `==` in a condition compares with a tiny epsilon, so `?x == 0.3` is safe for floats. `!=` also works. Comments start with `#`.

Dialog models with dialog_*

`dialog_new` makes a miniature neural language model: word embeddings feed a fixed window of preceding words into a dense network that predicts the next word. It is a few kilobytes and trains in a blink. It will not write a novel. It captures the cadence and vocabulary of the lines you feed it and samples new lines in that voice, so with a small corpus it mostly recites. That is a feature for game characters, whose entire personality is a handful of authored lines.

```
fn train_npc(seed) {
  bind npc = dialog_new(3, 16, 32, seed)
  dialog_learn(npc, "welcome traveler rest your boots by the fire")
  dialog_learn(npc, "welcome traveler the stew is hot and the ale is cold")
  dialog_learn(npc, "bandits hold the north pass keep your blade sharp")
  dialog_learn(npc, "the old mine woke something deep and hungry")
  dialog_train(npc, 60, 0.1)
  return npc
}

bind start = do {
  bind a = train_npc(1)
  bind b = train_npc(1)
  bind c = train_npc(2)
  println("greedy a:  ", dialog_say(a, "welcome traveler", 10, 0))
  println("greedy b:  ", dialog_say(b, "welcome traveler", 10, 0))
  println("same seed, same voice: ", dialog_say(a, "welcome traveler", 10, 0.8) ==
dialog_say(b, "welcome traveler", 10, 0.8))
  println("other seed: ", dialog_say(c, "welcome traveler", 10, 0))
  println("hot:       ", dialog_say(a, "bandits hold", 10, 1.5))
  println("untrained: [", dialog_say(dialog_new(3, 8, 8, 1), "hello", 5, 0), "]")
  println("bad handle: [", dialog_say(77, "hello", 5, 0), "]")
}
```

```
OUTPUT
greedy a:  the stew is hot and the ale is cold
greedy b:  the stew is hot and the ale is cold
same seed, same voice: true
other seed: rest your boots by the fire
hot:       the north pass keep your blade sharp
untrained: []
bad handle: []
```

The arguments to `dialog_new` are the context window (words of history), embedding width, hidden width and seed. `dialog_say(model, prompt, max_tokens, temperature)` seeds the window with the prompt and continues from there; the prompt is not echoed. Temperature 0

is greedy and stable. Higher temperatures sample, with top-k 40 and top-p 0.9. An untrained model returns an empty string and a bad handle does too, so test for `""` before displaying anything.

Training loss tells you whether it has learned the corpus. Watch it flatten:

```
bind start = do {
  bind npc = dialog_new(2, 12, 24, 5)
  dialog_learn(npc, "the harbour is quiet tonight")
  dialog_learn(npc, "the harbour bells ring at dawn")
  println("loss after 5 epochs:   ", round(dialog_train(npc, 5, 0.1) * 100) / 100)
  println("loss after 100 epochs: ", round(dialog_train(npc, 100, 0.1) * 100) / 100)
  println("loss after 400 epochs: ", round(dialog_train(npc, 400, 0.1) * 100) / 100)
  println("recites: ", dialog_say(npc, "the harbour", 8, 0))
  println("never saw this prompt: ", dialog_say(npc, "zebra quantum", 8, 0))
}
```

```
OUTPUT
loss after 5 epochs:   1.91
loss after 100 epochs: 0.14
loss after 400 epochs: 0.12
recites: bells ring at dawn
never saw this prompt: the harbour bells ring at dawn
```

A subtle point about randomness deserves attention, because it breaks naive tests. The model carries its own random generator, seeded at creation, and every sampled call advances it. Two calls with identical arguments can therefore differ, while the whole program still prints the same thing on every run:

```
bind start = do {
  bind npc = dialog_new(2, 12, 24, 4)
  dialog_learn(npc, "the road north is closed by snow")
  dialog_learn(npc, "the road south is open but the bridge is weak")
  dialog_learn(npc, "the road east leads to the harbour")
  dialog_train(npc, 200, 0.1)
  println("call 1: ", dialog_say(npc, "the road", 8, 1.2))
  println("call 2: ", dialog_say(npc, "the road", 8, 1.2))
  bind i = 0
  while i < 4 {
    println("temperature ", round((0.6 + i * 0.3) * 10) / 10, ": ", dialog_say(npc,
"the road", 8, 0.6 + i * 0.3))
    bind i = i + 1
  }
}
```

```
OUTPUT
call 1: east leads to the harbour
call 2: north is closed by snow
temperature 0.6: north is closed by snow
temperature 0.9: south is open but the bridge is weak
temperature 1.2: south is open but the bridge is weak
temperature 1.5: road is closed by snow
```

For repeatable tests use temperature 0, which does not depend on the generator, and build a fresh model per test. Exercise 3 turns this into a test.

For authored content, a corpus file is nicer than a pile of `dialog_learn` calls. The **Ling Dialog Standard** (`.lingdialog`) is plain text: `= label` starts a conversation, `speaker: text` is a turn, an indented line continues the previous turn, `#` starts a comment. `dialog_load(model, path)` adds a file and returns the number of conversations, or `-1` if it cannot be read. Writers can edit these files without touching code, and they diff cleanly in version control. `dialog_save` and `dialog_load_model` persist a trained model.

The dialog box

A separate family of `dialog_*` builtins draws the cinematic text box of a console adventure: typewriter reveal, colour-coded names, places and items, pages split by `||`. It is driven by your program one call per frame, and its state machine runs without a window, so the logic is

testable:

```
// The cinematic dialog box state machine, driven without a window.
bind start = do {
  dialog_show("Hey {n}Link{/}! Take the {i}Master Sword{/}.||Beware the night.", 20)
  println("active: ", dialog_active(), " typing: ", dialog_typing())
  bind frames = 0
  while dialog_typing() && frames < 400 {
    dialog_step(1 / 60)
    bind frames = frames + 1
  }
  println("page 1 revealed after ", frames, " frames (", round(frames / 60 * 10) / 10,
" s)")
  dialog_advance()
  println("after advance: active ", dialog_active(), " typing ", dialog_typing())
  bind frames = 0
  while dialog_typing() && frames < 400 { dialog_step(1 / 60) bind frames = frames + 1
}
  dialog_advance()
  println("closed: ", !dialog_active())
  dialog_show("skip me", 5)
  dialog_advance()
  println("advance while typing reveals: typing ", dialog_typing())
  dialog_close()
  println("after close: ", dialog_active())
}
```

```
OUTPUT
active: true typing: true
page 1 revealed after 96 frames (1.6 s)
after advance: active true typing true
closed: true
advance while typing reveals: typing false
after close: false
```



Figure: the same box on screen. Names, items and places use the markup `{n}...{/}`, `{i}...{/}`, `{p}...{/}`.

The first `dialog_advance` skipped nothing here because the typewriter had finished; had text still been typing, advance would have revealed the whole page first. In a game, gate your world update on `dialog_active()` so the player cannot walk away mid-sentence.

Mini-project: a sentry that decides, judges and speaks

Combine the three headless layers. A neural network learns to judge whether a fight is winnable from health, ammunition and the number of foes. A behaviour tree turns that confidence and the sighting of foes into an action. A dialog model, trained on three short lines, barks the matching line.

```
// sentry.ling — three small AIs cooperating, all headless.
fn lcg(s) { return (s * 75 + 74) % 65537 }
fn unit(s) { return s / 65537 }

// Ground truth the network must learn: fight when healthy, armed and not outnumbered.
fn should_fight(hp, ammo, foes) {
  if hp + ammo * 0.5 - foes * 0.25 > 0.6 { return 1 }
  return 0
}

fn train_threat() {
  bind net = nn_new(3, 11)
  nn_dense(net, 10, "tanh")
  nn_dense(net, 1, "sigmoid")
  bind s = 7
  bind i = 0
  while i < 20000 {
    bind s = lcg(s)
    bind hp = unit(s)
    bind s = lcg(s)
    bind ammo = unit(s)
    bind s = lcg(s)
    bind foes = floor(unit(s) * 5)
    nn_train(net, [hp, ammo, foes / 4], [should_fight(hp, ammo, foes)], 0.1)
    bind i = i + 1
  }
  return net
}

fn train_barks() {
  bind npc = dialog_new(2, 12, 24, 3)
  dialog_learn(npc, "attack for the crown and the king")
  dialog_learn(npc, "retreat fall back to the gate")
  dialog_learn(npc, "patrol all quiet on the wall")
  dialog_train(npc, 300, 0.1)
  return npc
}
```

```
bind start = do {
  bind threat = train_threat()
  bind barks = train_barks()
  bind brain = bt_build("selector {
    sequence { ?foes > 0 ?confidence >= 0.5 !attack }
    sequence { ?foes > 0 !retreat }
    !patrol
  }")
  // (hp, ammo, foes) over six moments of a siege
  bind moments = [[1.0, 0.9, 0], [0.9, 0.8, 1], [0.8, 0.7, 3], [0.4, 0.2, 2], [0.2,
0.1, 1], [1.0, 1.0, 0]]
  bind t = 0
  while t < len(moments) {
    bind m = moments[t]
    bind c = nn_forward(threat, [m[0], m[1], m[2] / 4])[0]
    bt_set(brain, "foes", m[2])
    bt_set(brain, "confidence", c)
    bind action = bt_tick(brain)
    bind line = dialog_say(barks, action, 6, 0)
    println("t=", t, " hp ", m[0], " ammo ", m[1], " foes ", m[2], " -> confidence ",
round(c * 100) / 100, " -> ", action, ": \"", line, "\"")
    bind t = t + 1
  }
}
```

```
OUTPUT
t=0 hp 1 ammo 0.9 foes 0 -> confidence 1 -> patrol: "all quiet on the wall"
t=1 hp 0.9 ammo 0.8 foes 1 -> confidence 1 -> attack: "for the crown and the king"
t=2 hp 0.8 ammo 0.7 foes 3 -> confidence 0.01 -> retreat: "fall back to the gate"
t=3 hp 0.4 ammo 0.2 foes 2 -> confidence 0 -> retreat: "fall back to the gate"
t=4 hp 0.2 ammo 0.1 foes 1 -> confidence 0 -> retreat: "fall back to the gate"
t=5 hp 1 ammo 1 foes 0 -> confidence 1 -> patrol: "all quiet on the wall"
```

Study how responsibility is divided. The network never chooses an action; it supplies one number, `confidence`, to the blackboard. The tree owns the decision and stays readable by a designer. The language model never decides anything; it dresses the decision in words. At t=2 the guard has good health but faces three foes, so confidence collapses to 0.01 and the tree

retreats instead of attacking, a call a hand-written threshold would take a dozen conditions to match. Because each layer is replaceable, you can swap the learned judgement for a rule, or the bark generator for authored lines, without touching the other two.

> **TRY IT**
>
> Change `should_fight` so that ammunition matters more than health, retrain, and rerun. Then add a fourth bark line for `investigate` and extend the tree with an `?alert > 0` branch.

Local language models: llm_ *and vision_*

The `llm_*` builtins run a chat model inside the `ling` process using a quantised GGUF checkpoint (Qwen3 or GLM-4 architectures) plus its `tokenizer.json`. `vision_*` answers questions about an image with a quantised Moondream2 model. Be clear about the requirements:

- A `ling` built with the Cargo feature `llm` (add `llm-cuda` for a GPU), and `vision` for images. They are off by default, and this book's reference build has neither.
- Model files that you supply. Nothing is downloaded for you.
- Memory for the whole model, loaded once.

Real inference could not be run while this chapter was written: the reference `ling` has no `llm` feature and no checkpoint was available. The examples below that load a model are therefore marked as model programs; they were checked against the reference documentation and the crate signatures, not executed. The probe and the fallback pattern *were* executed, in the build without the feature, which is exactly the situation you must handle.

Probe first. `llm_load` returns a handle, `-1` for a bad file in a capable build, and `()` in a build that lacks the feature:

```
// Is local LLM inference usable here? Three different answers are possible.
fn llm_status(gguf, tok) {
  bind m = llm_load(gguf, tok, 0 - 1)
  if m == () { return "not compiled in (build ling with --features llm)" }
  if m < 0 { return "compiled in, but the model or tokenizer could not be loaded" }
  return "ready"
}

bind start = do {
  println("llm:    ", llm_status("models/missing.gguf", "models/tokenizer.json"))
  bind v = vision_load("models/missing.gguf", "models/tok.json", 0 - 1)
  println("vision: ", v)
  println("job from an unknown model: [", llm_generate_start(5, "sys", "
<|user|>\nhi\n", 8, 0, 1), "]")
  println("done for an unknown job: ", llm_generate_done("nope"))
}
```

```
OUTPUT
llm:    not compiled in (build ling with --features llm)
vision: ()
job from an unknown model: [()]
done for an unknown job: ()
```

This is the three-way answer to build into any program: not compiled in, compiled in but no model, ready. Notice that `vision_load` and `llm_generate_start` evaluate to `()` as well, and the job functions return a harmless value for an unknown job instead of crashing.

> **NOTE**
>
> This probe depends on the JIT behaviour of unknown names. Run it under `ling run`, not `--interp`, which raises `unknown function 'llm_load'` at the first call in a build without the feature.

Design for the missing model

Because the large model is optional, hide it behind one function. The next program picks an engine once, then answers through a single `reply` function. With the feature absent, it falls back to the dialog model and the program still works:

```
// One entry point, two engines: the local LLM if present, otherwise the tiny dialog
model.
fn make_engine(gguf, tok, fallback) {
  bind m = llm_load(gguf, tok, 0 - 1)
  if m == () || m < 0 { return [0, fallback] }
  return [1, m]
}

fn reply(engine, system, question) {
  if engine[0] == 1 {
    bind job = llm_generate_start(engine[1], system, "<|user|>\n" + question + "\n",
96, 0.7, 0.9)
    while !llm_generate_done(job) { sleep(100) }
    return llm_generate_poll(job)
  }
  return dialog_say(engine[1], question, 8, 0)
}

bind start = do {
  bind small = dialog_new(2, 12, 24, 1)
  dialog_learn(small, "the shop opens at nine and closes at five")
  dialog_train(small, 300, 0.1)
  bind engine = make_engine("models/Qwen3-0.6B-Q8_0.gguf", "models/tokenizer.json",
small)
  println("engine: ", engine[0] == 1)
  println(reply(engine, "You are a shopkeeper. Be brief.", "the shop"))
}
```

```
OUTPUT
engine: false
opens at nine and closes at five
```

The real generation loop is the other branch. `llm_generate_start` returns a job id at once and decodes on a background thread; you poll for the text produced so far. History uses a fixed wire format: alternating `<|user|>` and `<|assistant|>` turns ending with a user turn. A system prompt is passed separately.

```
GUI
bind start = do {
  bind model = llm_load("models/Qwen3-0.6B-Q8_0.gguf", "models/tokenizer.json", 0 - 1)
  if model < 0 { println("could not load the model") return }
  bind history = "<|user|>\nWhy is the sky blue?\n"
  bind job = llm_generate_start(model, "Answer in one sentence.", history, 128, 0.7,
0.9)
  while !llm_generate_done(job) {
    print(llm_generate_poll(job))
    sleep(250)
  }
  println(llm_generate_poll(job))
}
```

The device argument is negative for the CPU and a CUDA index otherwise. Keep `max_tokens` modest: `llm_job_cancel` only marks the job done, the decode already running continues in the background until it reaches its limit, burning CPU or GPU the whole time. Tool calling is declared in the API (`llm_job_tool_name`, `llm_job_provide_tool_result`) but not implemented; no job ever reaches the `tool_call` status.

Images are sent as base64 text, since Ling strings cannot hold raw PNG bytes. A web page typically posts `canvas.toDataURL()` content to your handler; strip the `` prefix and pass the rest straight through. The program below reads such a base64 string from a text file.

```
GUI
bind start = do {
  bind model = vision_load("models/moondream2-q4.gguf", "models/moondream-
tokenizer.json", 0 - 1)
  bind job = vision_analyze_start(model, read_file("shot.b64"), "What is shown on the
screen?", 128, 0.2, 0.9)
  while !vision_analyze_done(job) { sleep(250) }
  if vision_analyze_error(job) == "" { println(vision_analyze_poll(job)) } else {
println("failed: ", vision_analyze_error(job)) }
}
```

> WARNING
>
> There is no moderation layer, and model output is untrusted text. Filter it before showing it to other users, and never hand it directly to file operations, shell commands or SQL. The same applies to the prompt: user input concatenated into a prompt can steer the model.

An HTTP front end that never blocks

The HTTP server is single-threaded, so a multi-second decode inside a handler would freeze every visitor. The job design exists to avoid that. Expose two routes: one starts a job and returns its id with status 202, the other reports progress. To keep the engine swappable, isolate it behind three functions. The version below uses the dialog model so it can be exercised on a laptop; to go live you rewrite only `engine_start`, `engine_poll` and `engine_done`.

GUI

```
// A chat endpoint with a swappable engine. The three engine_* functions are the
// only place that knows whether answers come from a dialog model or from llm_*.
form Response { status, content_type, body }

bind npc = dialog_new(2, 12, 24, 1)

fn engine_start(question) {
  bind n = to_number(shared_get("jobs", 0)) + 1
  shared_put("jobs", n)
  bind id = "job" + n
  shared_put(id, dialog_say(npc, question, 8, 0))
  return id
}
fn engine_poll(id) { return shared_get(id, "") }
fn engine_done(id) { return true }

fn json(status, body) { return Response(status, "application/json", body) }

fn ask(req) {
  bind q = trim(query_param(req.body, "q", ""))
  if q == "" { return json(400, "{\"error\":\"q is required\"}") }
  if len(q) > 500 { return json(413, "{\"error\":\"question too long\"}") }
  return json(202, "{\"job\":\"" + engine_start(q) + "\"}")
}

fn poll(req) {
  bind id = query_param(req.query, "job", "")
  return json(200, "{\"done\":" + engine_done(id) + ",\"text\":\"" +
json_escape(engine_poll(id)) + "\"}")
}

bind start = do {
  dialog_learn(npc, "the shop opens at nine and closes at five")
  dialog_train(npc, 300, 0.1)
  http_route("POST", "/ask", ask)
```

```
  http_route("GET", "/poll", poll)
  http_serve("127.0.0.1", to_number(cli_arg("port", "18141")))
}
```

Run it with `ling run chat_stub.ling --port 18141` and exercise it from a second terminal. This transcript is from a real run against `127.0.0.1`:

```
OUTPUT
$ curl -s -X POST -d 'q=the shop' http://127.0.0.1:18141/ask
{"job":"job1"}
$ curl -s -X POST -d 'q=the+shop+opens' http://127.0.0.1:18141/ask
{"job":"job2"}
$ curl -s "http://127.0.0.1:18141/poll?job=job2"
{"done":true,"text":"at nine and closes at five"}
$ curl -s "http://127.0.0.1:18141/poll?job=job1"
{"done":true,"text":"opens at nine and closes at five"}
$ curl -s -X POST -d 'q=' http://127.0.0.1:18141/ask
{"error":"q is required"}
```

Two details matter. The job counter lives in `shared_get` / `shared_put` because a `fn` cannot rebind a top-level variable; a counter kept as a plain `bind` would hand out `job1` forever. And `ling run` quietly uses the interpreter for any file containing `http_serve`, so the JIT probe trick from the previous section does not apply in a server: use an explicit configuration flag there. For the real engine, `engine_start` would call `llm_generate_start`, return its job id, and the other two would wrap `llm_generate_poll` and `llm_generate_done`.

The same engines in Rust: ling-ai

The builtins call the `ling-ai` crate. Depend on it directly:

```
[dependencies]
ling-ai = "2030"          # features = ["llm"] adds GGUF inference on candle
```

The crate edition is 2024, so your toolchain must be Rust 1.85 or newer.

Its extras are the `Tensor` type and the hooks for your own training loop. A tensor is an N-dimensional array of `f32` with `zeros`, `ones`, `from_vec`, `get`, `set`, element-wise `add` and `mul`, a naive 2-D `matmul`, `softmax` and `relu`. Shape mismatches panic. It is not a deep-

learning framework; it is the right size for small models you ship inside a game or tool.

The `Net` type is the one behind `nn_*`, but exposes `forward_cache` and `backward`, which let you supply any loss. Below, softmax cross-entropy is built from the tensor type: the gradient at the logits is `softmax - one_hot`. The network is trained on the circle task, evaluated, and written in the same `LNN1` format `nn_save` uses.

```rust
use ling_ai::{Activation, Net, Tensor};

// Tiny deterministic generator so the run is reproducible.
fn lcg(s: &mut u32) -> f32 {
    *s = (*s * 75 + 74) % 65537;
    *s as f32 / 65537.0 * 2.0 - 1.0
}

fn main() {
    // Tensors: shapes are checked, mismatches panic.
    let a = Tensor::from_vec(vec![2, 3], vec![1., 2., 3., 4., 5., 6.]);
    let b = Tensor::from_vec(vec![3, 2], vec![1., 0., 0., 1., 1., 1.]);
    let c = a.matmul(&b);
    println!("matmul {:?} -> {:?}", c.shape.0, c.data);
    let p = Tensor::from_vec(vec![3], vec![1., 2., 3.]).softmax();
    println!("softmax {:.4?} sums to {:.1}", p.data, p.data.iter().sum::<f32>());

    // A two-class classifier trained with our own loss (softmax cross-entropy)
    // by driving forward_cache and backward directly.
    let mut net = Net::new(2, 3);
    net.dense(12, Activation::Tanh).dense(8, Activation::Tanh).dense(2,
Activation::Linear);
    let mut seed = 1u32;
    for _ in 0..30_000 {
        let (x, y) = (lcg(&mut seed), lcg(&mut seed));
        let class = usize::from(x * x + y * y < 0.36);
        let acts = net.forward_cache(&[x, y]);
        let probs = Tensor::from_vec(vec![2], acts.last().unwrap().clone()).softmax();
        let mut grad = probs.data.clone();
        grad[class] -= 1.0; // d(cross-entropy)/d(logits) = softmax - one_hot
        net.backward(&acts, &grad, 0.05);
    }

    let mut right = 0;
    for _ in 0..1000 {
        let (x, y) = (lcg(&mut seed), lcg(&mut seed));
        let out = net.forward(&[x, y]);
        let guess = usize::from(out[1] > out[0]);
```

```
        right += usize::from(guess == usize::from(x * x + y * y < 0.36));
    }
    println!("held-out accuracy: {}%", right / 10);

    // Same LNN1 bytes the Ling builtins read and write.
    let bytes = net.to_bytes();
    std::fs::write("rust_circle.lnn", &bytes).unwrap();
    println!("wrote rust_circle.lnn ({} bytes); reload ok: {}", bytes.len(),
Net::from_bytes(&bytes).is_some());
    let logits = net.forward(&[0.0, 0.0]);
    println!("centre logits: {:.3?}", logits);
}
```

```
OUTPUT
matmul [2, 2] -> [4.0, 5.0, 10.0, 11.0]
softmax [0.0900, 0.2447, 0.6652] sums to 1.0
held-out accuracy: 96%
wrote rust_circle.lnn (671 bytes); reload ok: true
centre logits: [-4.086, 1.593]
```

Now the proof that the formats are shared. The file Rust just wrote loads in Ling and gives the same logits:

```
bind start = do {
  bind net = nn_load("rust_circle.lnn")
  println("loaded: ", net >= 0)
  bind out = nn_forward(net, [0, 0])
  println("centre logits: ", round(out[0] * 1000) / 1000, " ", round(out[1] * 1000) /
1000)
  bind far = nn_forward(net, [0.9, 0.9])
  println("corner says inside? ", far[1] > far[0])
}
```

```
OUTPUT
loaded: true
centre logits: -4.086 1.593
corner says inside? false
```

The same crate exposes the behaviour tree, dataset parser and dialog model, with the same semantics, including the unset-fact trap, which the first lines of this program sidestep by setting `hp` before the first tick:

```
use ling_ai::{ChatHistory, Dataset, DialogLM, LmConfig, Message, Tree};

fn main() {
    // Behaviour tree: the same text language as bt_build.
    let mut tree = Tree::parse("selector { sequence { ?hp < 0.3 !flee } !patrol
}").expect("valid tree");
    tree.set("hp", 1.0); // unset facts read as 0, which would trigger "flee"
    println!("tick 1: {}", tree.tick());
    tree.set("hp", 0.1);
    println!("tick 2: {} ({:?})", tree.tick(), tree.status());
    println!("hp reads back as {}", tree.get("hp"));
    println!("bad spec parses: {}", Tree::parse("selector {").is_some());

    // A .lingdialog corpus, parsed, then used to train the dialog model.
    let corpus = "@meta title: Tavern\n\n= welcome\nbram: welcome traveler rest your
boots by the fire\nhero: any news from the road\nbram: bandits on the north pass
nothing my stew cannot cure\n";
    let ds = Dataset::parse(corpus);
    println!("title {:?}, {} conversation(s), {} turns", ds.meta("title"),
ds.conversations.len(), ds.turn_count());
    let mut lm = DialogLM::new(LmConfig { ctx: 3, embed: 16, hidden: 32, seed: 1 });
    println!("training lines: {}", lm.learn_dataset(&ds));
    let loss = lm.train(40, 0.1);
    println!("loss under 2.0: {}", loss < 2.0);
    println!("npc says: {}", lm.say("welcome traveler", 8, 0.0));
    let bytes = lm.to_bytes();
    println!("model {} bytes, vocab {}", bytes.len(), lm.vocab_size());

    // Chat prompts for a real model (ChatML).
    let mut chat = ChatHistory::new();
    chat.push(Message::system("You are Bram, a tavern keeper."));
    chat.push(Message::user("Any rooms free?"));
    println!("{}", chat.to_chatml());
}
```

```
OUTPUT
tick 1: patrol
tick 2: flee (Success)
hp reads back as 0.1
bad spec parses: false
title Some("Tavern"), 1 conversation(s), 3 turns
training lines: 1
loss under 2.0: true
npc says: rest your boots by the fire
model 11443 bytes, vocab 25
<|im_start|>system
You are Bram, a tavern keeper.<|im_end|>
<|im_start|>user
Any rooms free?<|im_end|>
```

`DialogLM::say` takes `&mut self` because sampling advances the generator, the Rust-side reason behind the earlier `dl3.ling` output. `Dataset` also offers `to_lingdialog()` and `training_lines()`. `ChatHistory` and `Message` format conversations as plain text or ChatML for any chat model.

For real inference from Rust there are two engines. In `ling-ai` with the `llm` feature, `LoadedModel::load(gguf, tokenizer, device)` and `generate` run Qwen3 or GLM-4 through candle; that path needs the heavy candle dependencies and a model file, and was not built for this chapter. The second, `ling-llm`, is a freestanding `no_std` Qwen3 engine that reads the model, tokenizer and vocabulary out of one GGUF file and runs on the CPU, built so LingOS can answer a question with nothing else installed. Its signatures compile, and its prompt builder runs anywhere:

```rust
use ling_llm::Llm;

// Compiled but never called: it needs a Qwen3 GGUF file (about 600 MB for the 0.6B
Q8_0 model).
#[allow(dead_code)]
fn answer(model_bytes: &[u8]) -> Result<String, ling_llm::Error> {
    let llm = Llm::load(&model_bytes, &mut |done, total| eprintln!("{done}/{total}"))?;
    let mut state = llm.model.new_state(2048); // KV cache for up to 2048 tokens
    let mut out = String::new();
    llm.generate(
        &mut state,
        "You are concise.",                 // system prompt
        "<|user|>\nName three colours.\n",   // wire-format history
        64,                                  // max tokens
        0.0,                                 // temperature (0 = greedy)
        1.0,                                 // top-p
        1,                                   // seed
        &mut |piece| { out.push_str(piece); true }, // return false to stop early
    )?;
    Ok(out)
}

fn main() {
    // The prompt builder is plain text manipulation and runs anywhere.
    let prompt = ling_llm::chat::build_prompt("be brief", "<|user|>\nhi\n");
    println!("{prompt}");
}
```

OUTPUT

```
<|im_start|>system
be brief<|im_end|>
<|im_start|>user
hi<|im_end|>
<|im_start|>assistant
<think>

</think>
```

> TIP
>
> Run generation on a worker thread. `generate` holds the model lock and blocks until the reply is complete, the same reason the Ling builtins use jobs.

Chapter summary

- Pick the smallest layer that works: trees for decisions, networks for judgements, dialog models for voice, a local LLM only when you need open-ended language.
- Handles are integers; failures are soft (`-1`, `[]`, `""`). Create models once; handles are never reclaimed.
- `nn_train` returns the pre-update loss. Scale inputs, evaluate on unseen data, and treat `.lnn` files as untrusted input.
- Behaviour-tree facts default to 0, so initialise everything before ticking. `not` takes one child without braces; `bt_build` returns `-1` on a bad spec; check `bt_status` when using `parallel`.
- Dialog models recite small corpora by design. Sampling advances an internal generator, so use temperature 0 and fresh models in tests.
- `llm_*` and `vision_*` need a feature build and model files. Probe with `llm_load`, hide the engine behind a function, and keep HTTP handlers non-blocking with jobs.
- `ling-ai` shares file formats with the builtins; train in Rust, load in Ling, or the reverse.

Exercises

1. **Learn a curve.** Train a network to approximate `y = x * x` on `[-1, 1]` using only the builtins. Report the worst error over nine evenly spaced points, and what it predicts for `x = 2`.
2. **A shopkeeper.** Write a behaviour tree with four behaviours: serve when there are customers and stock, apologise when there are customers and no stock, lock up when it is closing time, otherwise sweep. Tick it through four situations.
3. **A dialog regression test.** Write a program that prints `PASS` or `FAIL` for four properties of a freshly built dialog model, including that greedy output is stable across calls and across rebuilds with the same seed.

Answers

1. The network needs a linear output layer and many small steps. The worst error on the grid is small, but `x = 2` is far outside the training range and the prediction is nowhere near 4: networks interpolate, they do not extrapolate.

```
fn lcg(s) { return (s * 75 + 74) % 65537 }

bind start = do {
  bind net = nn_new(1, 5)
  nn_dense(net, 10, "tanh")
  nn_dense(net, 1, "linear")
  bind s = 3
  bind i = 0
  while i < 40000 {
    bind s = lcg(s)
    bind x = s / 65537 * 2 - 1
    nn_train(net, [x], [x * x], 0.05)
    bind i = i + 1
  }
  bind worst = 0
  bind k = 0 - 4
  while k <= 4 {
    bind x = k / 4
    bind err = abs(nn_forward(net, [x])[0] - x * x)
    if err > worst { bind worst = err }
    bind k = k + 1
  }
  println("worst error on 9 points: ", round(worst * 1000) / 1000)
  println("f(0.5) = ", round(nn_forward(net, [0.5])[0] * 1000) / 1000, " (true 0.25)")
  println("f(2) = ", round(nn_forward(net, [2])[0] * 100) / 100, " (true 4: outside the
training range)")
}
```

```
OUTPUT
worst error on 9 points: 0.031
f(0.5) = 0.254 (true 0.25)
f(2) = 1.98 (true 4: outside the training range)
```

2. `not ?stock >= 1` turns the stock check into the second branch's guard, so the branches are mutually exclusive:

```
bind start = do {
  bind shop = bt_build("
    selector {
      sequence { ?customers > 0 ?stock >= 1 !serve }
      sequence { ?customers > 0 not ?stock >= 1 !apologise }
      sequence { ?closing > 0 !lock_up }
      !sweep
    }
  ")
  println("empty shop: ", bt_tick(shop))
  bt_set(shop, "customers", 2)
  bt_set(shop, "stock", 5)
  println("customers, stock: ", bt_tick(shop))
  bt_set(shop, "stock", 0)
  println("customers, no stock: ", bt_tick(shop))
  bt_set(shop, "customers", 0)
  bt_set(shop, "closing", 1)
  println("closing time: ", bt_tick(shop))
}
```

```
OUTPUT
empty shop: sweep
customers, stock: serve
customers, no stock: apologise
closing time: lock_up
```

3. A helper makes the intent readable. Temperature 0 makes the first two properties hold; the other two guard the empty-output cases:

```
fn make_npc() {
  bind npc = dialog_new(2, 12, 24, 4)
  dialog_learn(npc, "the road north is closed by snow")
  dialog_learn(npc, "the road south is open")
  dialog_train(npc, 200, 0.1)
  return npc
}

fn check(name, passed) {
  if passed { println("PASS ", name) } else { println("FAIL ", name) }
}

bind start = do {
  bind a = make_npc()
  bind first = dialog_say(a, "the road", 8, 0)
  check("greedy output repeats on the same model", dialog_say(a, "the road", 8, 0) ==
first)
  check("a rebuilt model with the same seed agrees", dialog_say(make_npc(), "the road",
8, 0) == first)
  check("untrained model says nothing", dialog_say(dialog_new(2, 4, 4, 1), "the road",
8, 0) == "")
  check("output is not empty", len(first) > 0)
}
```

OUTPUT

```
PASS greedy output repeats on the same model
PASS a rebuilt model with the same seed agrees
PASS untrained model says nothing
PASS output is not empty
```

CHAPTER 8

Using the Crates from Rust

What you'll learn

Every family of builtins in Ling is a thin, string-and-handle layer over an ordinary Rust crate. That means the code you have been calling from scripts is also a set of libraries you can put in a `Cargo.toml` and use from a Rust program, with no interpreter involved. This chapter is a guided tour. You will add Ling crates to a project and a workspace, run a short recipe for each of eleven crates, embed the interpreter itself to run scripts from a Rust application, and expose a builtin of your own.

Everything here was compiled and run: one Cargo project containing a binary per recipe, built against the crate sources in the Ling repository. Where a recipe needs hardware (a sound card, a window) the program either avoids it or compiles the device code without calling it, and says so. Output blocks are the real output of the real binaries.

You should be comfortable reading Rust. The chapter does not teach the language; it teaches the shape of the Ling crate family and the decisions you make when using it.

Why go native

Scripts are the right tool for iteration, content and glue. Rust is the right tool when you want the library without the language: a server that needs post-quantum cryptography, a build tool that wants Ling's vector fonts, a game written in Rust that wants the physics engine. Because the builtins wrap these crates one-to-one, knowing the script API means you already know most of the Rust API, and the examples in the reference chapters have a Rust twin.



The Ling workspace at a glance

Figure: the crate map. Every box in the middle layers is a library you can depend on directly.

Adding Ling crates to Cargo

The crates are published on crates.io under the `2030.x.y` version scheme (the HTTP crates `ling-http` and `ling-web` are `0.1.x`). A published dependency looks like this:

```
[dependencies]
ling-crypto = "2030"
ling-ai = "2030"
ling-physics = "2030.1"
```

From a checkout of the repository, use a path instead. A path dependency on a single crate works without copying the whole workspace, because Cargo resolves the workspace inheritance for you:

```
[dependencies]
ling-crypto = { path = "../ling/crates/ling-crypto" }
```

Every recipe in this chapter was built with path dependencies, so the version numbers below are those in the checkout the recipes ran against. Pin the one you tested with:

Crate	Version in the checkout	Imported as	Edition
`ling-audio`	2030.0.13	`ling_audio`	2021
`ling-music`	2030.0.9	`ling_music`	2021
`ling-physics`	2030.1.18	`ling_physics`	2021
`ling-game`	2030.1.9	`ling_game`	2021
`ling-graphics`	2030.0.13	`ling_graphics`	2021
`ling-ui`	2030.0.9	`ling_ui`	2021
`ling-animation`	2030.0.8	`ling_animation`	2021
`ling-input`	2030.0.9	`ling_input`	2021
`ling-crypto`	2030.1.13	`ling_crypto`	2021
`ling-net`	2030.0.9	`ling_net`	2021
`ling-ai`	2030.0.10	`ling_ai`	2024
`ling-fu`	2030.1.19	`lingfu` (package name `lingfu`)	2024
`ling-lang` (root)	2030.1.53	`ling`	2021

Two crates are on edition 2024, so a consuming project needs Rust 1.85 or newer. Note the two naming oddities: the project manager crate in the directory `ling-fu` is the package `lingfu`, and the root crate `ling-lang` is imported as `ling`.

Features matter. `ling-net` defaults to `http` (axum and reqwest); the post-quantum `lingtp` transport is a separate feature and so are `websocket`, `quic`, `grpc` and `lwp`. `ling-crypto` is `no_std`-capable with `--no-default-features --features alloc` once you install an entropy source. `ling-ai` adds GGUF inference with `llm` and `vision`. `ling-input` takes `gilrs` for native gamepads, `ling-physics` takes `cuda`. Enable only what you use; some of these pull in hundreds of crates.

One workspace, one place for versions

When your own code is split into crates, pin the Ling dependencies once in `[workspace.dependencies]` and inherit them. This is a complete, building example: a library that derives a key from a passphrase and seals a note, and a binary that uses it.

```
[workspace]
resolver = "2"
members = ["vaultlib", "vaultcli"]

[workspace.package]
edition = "2021"
version = "0.1.0"

# One place to pin every ling crate. Use version = "2030.1" from crates.io;
# the path form below is the checkout form used while testing this chapter.
[workspace.dependencies]
ling-crypto = { path = "C:/Users/User/Programs/ling/crates/ling-crypto" }
ling-ai     = { path = "C:/Users/User/Programs/ling/crates/ling-ai" }
```

```
use ling_crypto::lw::kdf::{password_key, PasswordParams};
use ling_crypto::XChaCha20;

/// Seal a note under a passphrase. Output: salt (16) || nonce || ciphertext || tag.
pub fn seal_note(passphrase: &str, salt: [u8; 16], note: &str) -> Vec<u8> {
    let params = PasswordParams { m_cost_kib: 4096, t_cost: 2, p_cost: 1 };
    let key = password_key(passphrase.as_bytes(), &salt, params).expect("kdf");
    let mut out = salt.to_vec();
    out.extend(XChaCha20::new(*key).encrypt(note.as_bytes()).expect("seal"));
    out
}

pub fn open_note(passphrase: &str, blob: &[u8]) -> Result<String, &'static str> {
    let (salt, rest) = blob.split_at(16);
    let params = PasswordParams { m_cost_kib: 4096, t_cost: 2, p_cost: 1 };
    let key = password_key(passphrase.as_bytes(), salt.try_into().unwrap(),
params).map_err(|_| "kdf")?;
    let plain = XChaCha20::new(*key).decrypt(rest)?;
    String::from_utf8(plain).map_err(|_| "not utf-8")
}
```

```
fn main() {
    let blob = vaultlib::seal_note("correct horse", [7u8; 16], "meeting moved to 3pm");
    println!("blob is {} bytes", blob.len());
    println!("{:?}", vaultlib::open_note("correct horse", &blob));
    println!("{:?}", vaultlib::open_note("wrong horse", &blob));
}
```

```
OUTPUT
blob is 76 bytes
Ok("meeting moved to 3pm")
Err("decryption/auth failed")
```

The `salt` argument is explicit so the output is repeatable; a real caller passes 16 random bytes. The blob is salt (16) plus nonce (24) plus 20 bytes of text plus a 16-byte tag, 76 bytes.

> **TIP**
>
> Windows has a 260-character path limit that bites deep `target` directories. While preparing this chapter a link step failed with `LNK1104: cannot open file` for an object that plainly existed, because the build directory sat in a long path. Point `CARGO_TARGET_DIR` at a short path such as `C:\t` and the problem disappears.

> **NOTE**
>
> The first build compiles a large dependency tree and takes minutes; later builds are incremental. Share one `CARGO_TARGET_DIR` between related projects so the Ling crates compile once.

A recipe per crate

Each recipe is a complete `main.rs`. They are chosen to run headless and deterministically so you can paste them into a test.

`ling-audio`: signals and sound

`AudioEngine` opens the default output device and plays tones, one-shots and samples. `FftAnalyzer` is pure DSP and needs no device, which makes it the testable half. The `beep` function below compiles the device path without calling it; on a machine with speakers you would call it.

```rust
use ling_audio::{AudioEngine, FftAnalyzer, ToneParams, Wave};

// Needs an output device: compiled here, called only on a machine with speakers.
#[allow(dead_code)]
fn beep() -> Result<(), Box<dyn std::error::Error>> {
    let audio = AudioEngine::new()?;
    audio.set_tone(0, ToneParams { x: 2.0, z: 6.0, freq: 196.0, ..Default::default()
});
    audio.blip(880.0, 0.25, 0.06, Wave::Square);
    Ok(())
}

fn main() {
    // The analyser is pure DSP: feed it a 1 kHz sine and ask what it hears.
    let mut fft = FftAnalyzer::new(2048, 44_100);
    let sine: Vec<f32> = (0..2048)
        .map(|i| (std::f32::consts::TAU * 1000.0 * i as f32 / 44_100.0).sin())
        .collect();
    fft.push_samples(&sine);
    println!("dominant: {:.0} Hz, {} bands", fft.dominant_freq(),
fft.freq_bands(8).len());
}
```

```
OUTPUT
dominant: 991 Hz, 8 bands
```

The 991 Hz answer is the centre of the FFT bin nearest 1000 Hz; bins here are about 21.5 Hz wide.

ling-music: synthesis and analysis

Decode audio, detect tempo and key, synthesise notes from small patch files, read MIDI. This recipe synthesises half a second of A4 and then recovers its pitch, exercising both directions with no device.

```
use ling_music::{note, patch, pitch, Synth};

fn main() {
    let text = r#"
        name = "Pure"
        op1.wave=sine op1.level=1.0 op1.adsr=0.005,0.1,0.8,0.2
        amp = 0.8
    "#;
    let mut synth = Synth::new(44_100.0);
    let id = synth.add_patch(patch::from_str(text));
    synth.note(id, 69, 0.9, 0.5); // A4 for half a second
    let samples: Vec<f32> = (0..22_050).map(|_| synth.next_sample()).collect();
    let hz = pitch::detect(&samples[4_000..8_096], 44_100).expect("pitch");
    println!("synthesised {:.1} Hz = {}", hz, note::hz_to_name(hz));
    println!("C4 is MIDI {:?}", note::name_to_midi("C4"));
}
```

```
OUTPUT
synthesised 440.0 Hz = A4
C4 is MIDI Some(60)
```

ling-physics: bodies, liquids, hyperbolic space

Rigid and soft bodies, liquids, terrain and weather. In this API `y` is up for the rigid world; mind the sign conventions in each module.

```
use ling_physics::hyperbolic::distance;
use ling_physics::rigid::{PhysicsWorld, RigidBody};
use ling_physics::Vec3;

fn main() {
    let mut world = PhysicsWorld::new();
    let id = world.add(RigidBody::new(Vec3::new(0.0, 5.0, 0.0), 1.0));
    for _ in 0..240 {
        world.step(0.016);
        world.bodies[id].bounce_floor(0.0, 0.5, 0.6); // floor y, restitution, friction
    }
    println!("box rests at y = {:.2}", world.bodies[id].pos.y);
    println!("hyperbolic d(0, 0.99) = {:.3}", distance(Vec3::ZERO, Vec3::new(0.99, 0.0,
0.0)));
}
```

```
OUTPUT
box rests at y = 0.50
hyperbolic d(0, 0.99) = 5.293
```

A box dropped from five metres settles at y = 0.5, half its height above the floor; the hyperbolic distance grows without bound near the boundary of the unit ball.

ling-game: ECS, 2D physics, meshes, dialog

Sparse-set entity storage, 2D rigid bodies and collision helpers, procedural meshes and the typewriter dialog state used by the `dialog_*` builtins.

```
use ling_game::entity::{ComponentStore, Entity};
use ling_game::physics::Aabb;
use ling_game::{mesh, Dialog};

struct Health(i32);

fn main() {
    let mut ids = Entity::new();
    let mut health: ComponentStore<Health> = ComponentStore::new();
    let hero = ids.spawn();
    health.insert(hero, Health(10));
    health.get_mut(hero).unwrap().0 -= 3;
    println!("hero hp {}", health.get(hero).unwrap().0);

    println!("overlap: {}", Aabb::new(0.0, 0.0, 1.0, 1.0).overlaps(Aabb::new(1.5, 0.0,
1.0, 1.0)));
    let ball = mesh::sphere(1.0, 8, 12);
    println!("sphere: {} vertices, {} triangles", ball.vertices.len(),
ball.indices.len() / 3);

    let mut d = Dialog::new("Hi {n}Ling{/}||Bye", 100.0);
    d.update(0.5);
    println!("still typing after 0.5 s: {}", d.is_typing());
}
```

```
OUTPUT
hero hp 7
overlap: true
sphere: 117 vertices, 192 triangles
still typing after 0.5 s: false
```

ling-graphics: a software renderer

A scene graph, cameras, materials and a CPU rasteriser that renders into an RGBA framebuffer in memory, so it needs no window. Here the framebuffer is mapped to ASCII by brightness. The same framebuffer could be written out as an image or uploaded to a texture.

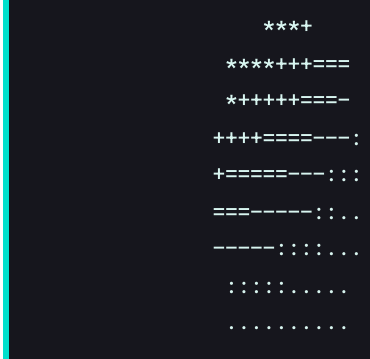
```
use ling_graphics::*;
use std::sync::Arc;

fn main() {
    let mut scene = Scene::new();
    let mesh = Arc::new(geometry::sphere(0.8, 16, 24));
    let material = Arc::new(Material::new(Color::from_srgb(230, 120, 90)));
    let place = scene::Transform::from_translation(Vec3::new(0.0, 0.0, -2.2));
    scene.spawn_mesh("ball", mesh, material, place);

    let camera = Camera3D::perspective(60.0, 2.0, 0.1, 100.0);
    let mut window = window::Window::new(viewport::Viewport::new(40, 16,
Color::BLACK));
    let fb = window.render_scene(&scene, &camera);

    let ramp: Vec<char> = " .:-=+*#%@".chars().collect();
    for row in fb.pixels.chunks(fb.width as usize * 4) {
        let line: String = row
            .chunks(4)
            .map(|p| ramp[(p[0].max(p[1]).max(p[2]) as usize * (ramp.len() - 1)) /
255])
            .collect();
        println!("{line}");
    }
}
```

OUTPUT



ling-ui: widgets as geometry

`ling-ui` does not draw. Widgets, flexbox-style layout, springs and tweens return resolved geometry and numbers, and you render them with whatever you have. That separation is why they test so easily.

```
use ling_ui::*;

fn main() {
    // Widgets return resolved geometry; you decide how to draw it.
    let bar = widgets::bar(20.0, 20.0, 200.0, 14.0, 0.7, 0x00d2ff, 0x2c3e64);
    println!("bar: {} fill(s), {} stroke(s)", bar.fills.len(), bar.strokes.len());

    let parent = LayoutNode { x: 0.0, y: 0.0, width: 300.0, height: 100.0 };
    let style = Style { direction: FlexDirection::Row, gap: 10.0, padding: 5.0 };
    for (i, n) in simple_layout(parent, 3, &style).iter().enumerate() {
        println!("child {i}: x {:.0} w {:.0}", n.x, n.width);
    }

    let mut spring = Spring::ui(0.0);
    spring.set_target(1.0);
    for _ in 0..60 { spring.update(1.0 / 60.0); }
    println!("spring after 1 s: {:.3}, settled: {}", spring.value,
spring.is_settled());
}
```

```
OUTPUT
bar: 1 fill(s), 1 stroke(s)
child 0: x 5 w 90
child 1: x 105 w 90
child 2: x 205 w 90
spring after 1 s: 1.000, settled: true
```

ling-animation: tracks, timelines, temperament

Keyframe tracks with per-key easing, a clock, procedural gaits, mechanisms and facial rigs. `Temperament` blends between a rigid and a lively response.

```
use ling_animation::{EaseFunction, Temperament, Timeline, Track};

fn main() {
    let track = Track::new()
        .key(0.0, 0.0f32, EaseFunction::Linear)
        .key(1.0, 100.0, EaseFunction::CubicOut)
        .key(2.0, 40.0, EaseFunction::BounceOut);
    let mut clock = Timeline::new(track.duration());
    for _ in 0..3 {
        let t = clock.tick(0.5);
        println!("t = {:.1}s  value = {:.2}", t, track.sample(t).unwrap());
    }
    let mood = Temperament::new(0.25);
    println!("liveliness {:.2}, blend(10, 20) = {:.1}", mood.liveliness(),
mood.blend(10.0, 20.0));
}
```

```
OUTPUT
t = 0.5s  value = 50.00
t = 1.0s  value = 100.00
t = 1.5s  value = 47.50
liveliness 0.75, blend(10, 20) = 12.5
```

ling-input: devices and action maps

`Sensorium` aggregates gamepads, touch and XR devices; an action map turns physical buttons into named actions such as `jump`. Real devices come from a backend (`gilrs` for pads); here the event is injected by hand, which is exactly how you test input code.

```rust
use ling_input::{
    Action, ActionMap, ActionSet, Binding, DeviceInfo, DeviceKind, GamepadButton,
InputEvent,
    Sensorium,
};

fn main() {
    let mut s = Sensorium::new(2);

    // Register a pad by hand (a real backend such as GilrsBackend does this for you).
    let id = s.devices.add(DeviceInfo::new(0, DeviceKind::Gamepad, "Test pad"));
    s.devices.assign_player(id);

    // Describe "jump" as an action bound to the bottom face button.
    let mut gameplay = ActionMap::new();
    gameplay.set("jump", Action::new().bind(Binding::Pad(GamepadButton::South)));
    s.actions.define("gameplay", ActionSet { map: gameplay, blocking: false });
    s.actions.push("gameplay");

    // One frame: the button goes down.
    s.begin_frame();
    s.ingest(InputEvent::Button { device: id, button: GamepadButton::South, pressed:
true });
    s.update(1.0 / 60.0);

    let jump = s.actions.map("gameplay").unwrap().just_pressed("jump");
    println!("jump just pressed: {jump}");
    println!("south is down: {}", s.player(0).unwrap().is_down(GamepadButton::South));
}
```

```
OUTPUT
jump just pressed: true
south is down: true
```

ling-crypto: hashing, sealing, signing

The same primitives as the previous chapter in a dozen lines. `Blake3::hash` matches `hash_hex`, and `XChaCha20` output uses the `nonce || ciphertext || tag` layout that `crypto_seal` produces.

```
use ling_crypto::{Blake3, Ed25519Keypair, XChaCha20};

fn main() {
    let digest = Blake3::hash(b"abc");
    println!("blake3 starts {:02x}{:02x}{:02x}{:02x}", digest[0], digest[1], digest[2],
digest[3]);

    let cipher = XChaCha20::new(XChaCha20::generate_key());
    let sealed = cipher.encrypt(b"attack at dawn").unwrap();
    println!("sealed {} bytes, opens: {:?}", sealed.len(),
cipher.decrypt(&sealed).map(|p| p.len()));

    let kp = Ed25519Keypair::generate();
    let sig = kp.sign(b"msg");
    println!("signature ok: {}", Ed25519Keypair::verify(&kp.public_key(), b"msg",
&sig).is_ok());
    println!("wrong message ok: {}", Ed25519Keypair::verify(&kp.public_key(), b"other",
&sig).is_ok());
}
```

```
OUTPUT
blake3 starts 6437b3ac
sealed 54 bytes, opens: Ok(14)
signature ok: true
wrong message ok: false
```

ling-net: the lingtp transport

`ling-net` with the `lingtp` feature is a post-quantum, raw-TCP transport: an ML-DSA-87 server identity, a hybrid KEM handshake, and request/response framing. This server and client run on loopback in one process. Note that the Ling runtime and `ling-net` have two different lingtp wire formats, so a Rust client talks to a Rust server here, not to `lingtp_serve` in a script.

```
use ling_crypto::MlDsa87Keypair;
use ling_net::lingtp::{client, server, LingtpRequest, LingtpResponse};
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:0").unwrap();
    let port = listener.local_addr().unwrap().port();
    let identity = MlDsa87Keypair::generate();
    std::thread::spawn(move || {
        let _ = server::serve_on(listener, identity, |req| {
            if req.path == "/health" {
                LingtpResponse::text(200, "ok")
            } else {
                LingtpResponse::error(404, "not found")
            }
        });
    });
    let mut conn = client::connect("127.0.0.1", port).unwrap();
    for path in ["/health", "/missing"] {
        let r = conn.request(&LingtpRequest::get(path)).unwrap();
        println!("GET {path} -> {} {}", r.status, r.body);
    }
}
```

```
OUTPUT
GET /health -> 200 ok
GET /missing -> 404 {"error":"not found"}
```

ling-fu: the project manager as a library

The `lingfu` crate exposes its keyword normaliser, so a Rust tool can rewrite Ling source into another script. This is the same code behind `lingfu normalize`.

```
use lingfu::normalize::{normalize_content, Lang};

fn main() {
    let src = "fn fib(n) {\n  if n < 2 { return n }\n  return fib(n - 1) + fib(n -
2)\n}\n";
    for code in ["zh", "ko", "ru"] {
        let lang = Lang::from_name(code).expect("known language");
        println!("--- {} ---", lang.name());
        print!("{}", normalize_content(src, lang));
    }
}
```

OUTPUT

```
--- Chinese ---
函 fib(n) {
  若 n < 2 { 归 n }
  归 fib(n - 1) + fib(n - 2)
}
--- Korean ---
함수 fib(n) {
  만약 n < 2 { 반환 n }
  반환 fib(n - 1) + fib(n - 2)
}
--- Russian ---
функция fib(n) {
  если n < 2 { вернуть n }
  вернуть fib(n - 1) + fib(n - 2)
}
```

WARNING

Normalising back to English from Thai or other targets can leave words untranslated. Keep an English original in version control and treat normalisation as one-way.

Embedding ling-lang

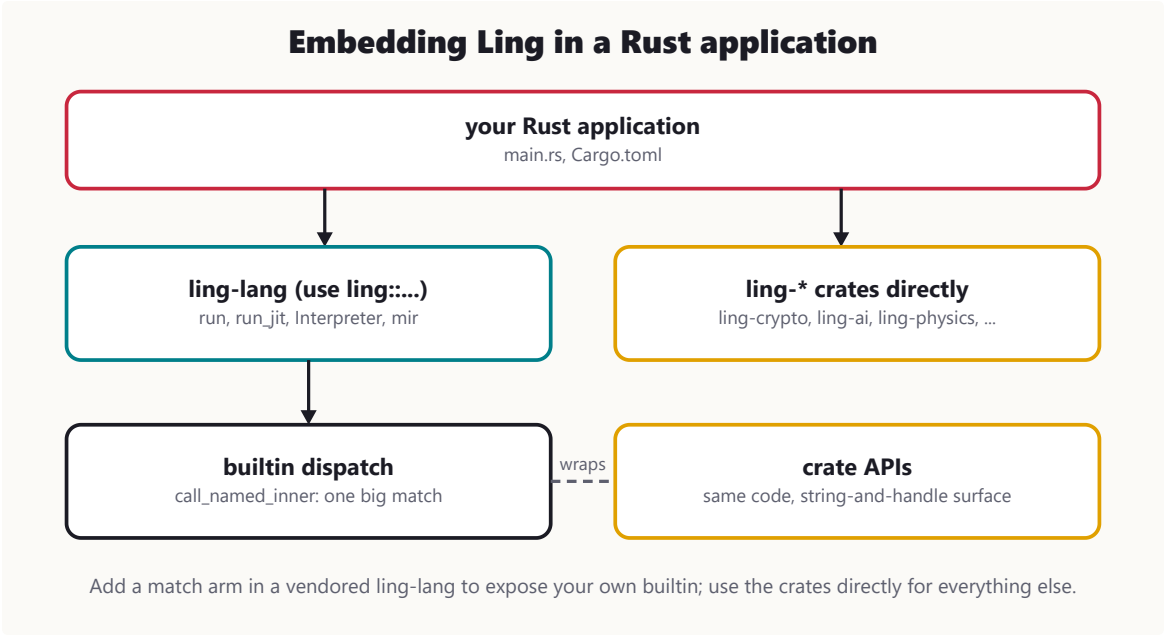


Figure: two routes from a Rust program. Depend on `ling-lang` to run scripts through the interpreter, or depend on the individual crates and skip the language entirely.

The root crate is both a program and a library. Depend on `ling-lang` and import it as `ling`; `ling::run(source)` lexes, parses and runs a script's entry binding. `ling::run_jit(source, dir, name)` uses the Cranelift path. Errors return as a fully rendered diagnostic string rather than a panic, so a host can show or log them.

```
fn main() {
    let src = r#"bind start = do { println("hello from ", "an embedded interpreter")
}"#;

    // One call: lex, parse, interpret, run the entry binding.
    ling::run(src).expect("script failed");

    // The Cranelift JIT path.
    ling::run_jit(src, None, Some("embedded.ling")).expect("jit failed");

    // Errors come back as rendered diagnostics, not panics.
    let err = ling::run(r#"bind start = do { println(1 + ) }"#).unwrap_err();
    println!("first line of the diagnostic: {}", err.lines().next().unwrap_or(""));
}
```

```
OUTPUT
hello from an embedded interpreter
hello from an embedded interpreter
first line of the diagnostic: error[parse]: unexpected token in expression: RParen
```

The first two lines are the script's own output from the interpreter and the JIT. The third shows an error returned as text. For finer control, `ling::parser::parse` gives you a `Program` you can run on an `Interpreter::new()` you own, and `ling::mir::compile_and_optimize` returns the optimised MIR to inspect.

The next program runs several scripts in a loop and reports each result, which is the shape of a plugin host:

```
fn main() {
    let scripts = [
        ("good", r#"bind start = do { println("sum is ", 2 + 3) }"#),
        ("broken", r#"bind start = do { println(1 + ) }"#),
        ("empty", "// nothing here"),
    ];
    for (name, src) in scripts {
        match ling::run(src) {
            Ok(()) => println!("{name}: ok"),
            Err(e) => println!("{name}: failed with {}",
e.lines().next().unwrap_or("")),
        }
    }
}
```

```
OUTPUT
sum is 5
good: ok
broken: failed with error[parse]: unexpected token in expression: RParen
empty: failed with error[runtime]: no entry point — need `bind start = do {...}` or `ผูก
เริ่ม = ทำ {...}`
```

Notice the third script. A file with no entry binding is an error (`no entry point`), not a silent success.

Three cautions for production. Embedding pulls in the whole runtime and its windowing and audio dependencies; there is no slim "language only" crate, because `ling-lex`, `ling-ast` and `ling-polyglot` are not wired into the interpreter. Process-wide state is shared: the network, SSH, AI and video builtins keep handle tables in statics, so two interpreters in one process see the same handles. And a script can do anything its builtins can: read files, open sockets, spawn processes. Untrusted scripts need isolates with explicit capabilities (see *Concurrency and Isolates*), not just `ling::run`.

Exposing your own builtin

As the embedding figure shows, the interpreter dispatches every builtin through one `match` in `Interpreter::call_named_inner`, in `src/runtime/mod.rs`. There is no public registration hook today: to add a builtin you add an arm to that match in a copy of `ling-lang` and depend on your copy. The JIT calls the same function, so one arm serves both backends.

The workflow is to vendor the root crate (its `src/`, `build.rs`, `lexicons/`, `stt/` and the icon), point its path dependencies at the shared `crates/` directory, and insert your arms before an existing one such as `"str_upper"`. Arguments are read with `arg_str`, `arg_num` and `arg_list_str`, which supply defaults for missing arguments. A builtin returns `Value::Number`, `Value::Str`, `Value::Unit`, or a result built with `Value::Ok(Box::new(..))` and `Value::Err(Box::new(..))`, which scripts destructure with `ok(v)` and `bad(e)`. These two arms were added to the vendored copy used for this chapter:

```
// inside Interpreter::call_named_inner, before the "str_upper" arm
"word_count" => {
    let s = self.arg_str(&args, 0, "");
    return Ok(Value::Number(s.split_whitespace().count() as f64));
},
"parse_port" => {
    let s = self.arg_str(&args, 0, "");
    return Ok(match s.trim().parse::<u16>() {
        Ok(p) if p >= 1024 => Value::Ok(Box::new(Value::Number(p as f64))),
        Ok(_) => Value::Err(Box::new(Value::Str("port is privileged".into()))),
        Err(_) => Value::Err(Box::new(Value::Str(format!("not a port: {s}")))),
    });
},
```

That fragment is not a standalone program; it compiled inside the vendored crate. The host that uses the patched crate is ordinary:

```
fn main() {
    let script = r#"
        bind start = do {
          println("words: ", word_count("the quick brown fox"))
          println(parse_port("8080"))
          println(parse_port("80"))
          println(parse_port("http"))
          match parse_port("9000") {
            ok(p) => println("listening on ", p),
            bad(e) => println("error: ", e)
          }
        }
    "#;
    ling::run(script).expect("script failed");
}
```

```
OUTPUT
words: 4
Ok(8080)
Err(port is privileged)
Err(not a port: http)
listening on 9000
```

Both builtins work like any other, including the result convention in the `match`. A misspelt builtin would have evaluated to `()` silently, so add a test for every one you create.

WARNING

A vendored fork drifts from upstream. Keep your arms in one clearly marked block, keep the patch as a file you can reapply, and write a script test per builtin so an upgrade tells you what broke. If a capability is wanted by many users, propose it upstream instead.

TRY IT

If your custom builtin needs state that outlives a call (a database connection, a model), follow the pattern the runtime uses for networking and AI: a process-wide `static` table indexed by integer handles that fails softly on a bad handle.

Chapter summary

- Builtins wrap crates one-to-one; the Rust API mirrors the script API and you can use either side of the family alone.
- Use `2030` version requirements from crates.io or path dependencies from a checkout. `ling-ai` and `ling-fu` are edition 2024 (Rust 1.85+). Pin versions in `[workspace.dependencies]`.
- Enable only the features you need: `ling-net` has `lingtp`, `websocket`, `quic`, `grpc`, `lwp`; `ling-ai` has `llm`, `vision`; `ling-input` has `gilrs`.
- Prefer headless, deterministic recipes (`FftAnalyzer`, software rendering, injected input events) so Ling-backed code is unit-testable.
- `ling-lang` is imported as `ling`: `run`, `run_jit`, `parser::parse` plus `Interpreter`, `mir::compile_and_optimize`. Errors come back as rendered strings.
- Embedded interpreters share process-wide handle tables and have full builtin power; sandbox untrusted scripts with isolates.
- A custom builtin is a match arm in a vendored `ling-lang`, serving both interpreter and JIT.
- On Windows, keep the Cargo target path short.

Exercises

1. **Hash a file twice.** Write a Rust binary that depends only on `ling-crypto`, takes a path argument and prints the BLAKE3 digest of the file. Prove it equals Ling's `file_hash` on the same file.
2. **A script host.** Embed `ling-lang` and run a list of three scripts (one good, one with a syntax error, one empty), printing one result line per script.
3. **A third builtin.** In the vendored crate, add `title_case(text)` that capitalises each word and lowercases the rest, and call it from a script through an embedded interpreter.

Answers

1. Read the bytes and hash them. Ling's `file_hash` hashes raw bytes too, so the digests agree. First the Ling side writes a file and prints its hash:

```
bind start = do {
  write_file("sample.txt", "hello ling\n")
  println(file_hash("sample.txt"))
}
```

```
OUTPUT
a75a765aadefc9851024f166c79bd2157146cd4b330452f8f7d7e7039d65f1fb
```

Then the Rust side, run on the same file:

```
use ling_crypto::Blake3;

fn main() {
    let path = std::env::args().nth(1).expect("usage: ex_hash <file>");
    let bytes = std::fs::read(&path).expect("readable file");
    let digest: String = Blake3::hash(&bytes).iter().map(|b| format!("
{b:02x}")).collect();
    println!("{digest}");
}
```

```
OUTPUT
a75a765aadefc9851024f166c79bd2157146cd4b330452f8f7d7e7039d65f1fb
```

2. `ling::run` returns `Err(String)` with the diagnostic; the first line is the headline. The empty script is an error because it has no entry binding:

```
fn main() {
    let scripts = [
        ("good", r#"bind start = do { println("sum is ", 2 + 3) }"#),
        ("broken", r#"bind start = do { println(1 + ) }"#),
        ("empty", "// nothing here"),
    ];
    for (name, src) in scripts {
        match ling::run(src) {
            Ok(()) => println!("{name}: ok"),
            Err(e) => println!("{name}: failed with {}",
e.lines().next().unwrap_or("")),
        }
    }
}
```

```
OUTPUT
sum is 5
good: ok
broken: failed with error[parse]: unexpected token in expression: RParen
empty: failed with error[runtime]: no entry point — need `bind start = do {...}` or `ผูก
เริ่ม = ทำ {...}`
```

3. Add this arm next to the others, rebuild the vendored crate, and call it from a script. Unicode-aware `to_uppercase` on the first character and `to_lowercase` on the rest does the work:

```
"title_case" => {
    let s = self.arg_str(&args, 0, "");
    let out: Vec<String> = s
        .split_whitespace()
        .map(|w| {
            let mut cs = w.chars();
            match cs.next() {
                Some(c) => c.to_uppercase().collect::<String>() +
&cs.as_str().to_lowercase(),
                None => String::new(),
            }
        })
        .collect();
    return Ok(Value::Str(out.join(" ")));
},
```

The host program and its output:

```
fn main() {
    ling::run(r#"bind start = do { println(title_case("the QUICK brown fox"))
}"#).unwrap();
}
```

```
OUTPUT
The Quick Brown Fox
```

CHAPTER 9

Extending Ling

What you'll learn

Sooner or later a professional project hits the edge of a language: a builtin you wish existed, a vocabulary your team would rather write in, an editor that does not know your helper functions. Ling is small enough that you can change all of these yourself, and this chapter shows how, using only things that were built and run for this book.

You will trace exactly how a builtin call travels from your program to Rust, add a new builtin (`wrap`) to a copy of the runtime and watch it work on both backends, teach the lexer a new vocabulary (Latin), and learn the cheaper alternative of a user-level lexicon written in Ling itself. You will also see which parts of the workspace are real extension points and which are placeholders, so you do not spend a week patching a crate that the `ling` binary never calls. Finally we look at the VS Code extension, the Python bindings and what a contribution to the repository involves.

Prerequisites: you have read *How Ling Runs* (the pipeline and the two backends) and you can build a Rust project. Rust sections below were compiled; the Ling programs were run with `ling run` (JIT) and `ling run --interp` unless stated.

The map: what you can actually extend

The first lesson of extension work is to find the code that is really on the execution path. The workspace has many crates, and the `ling` binary uses only some of them.

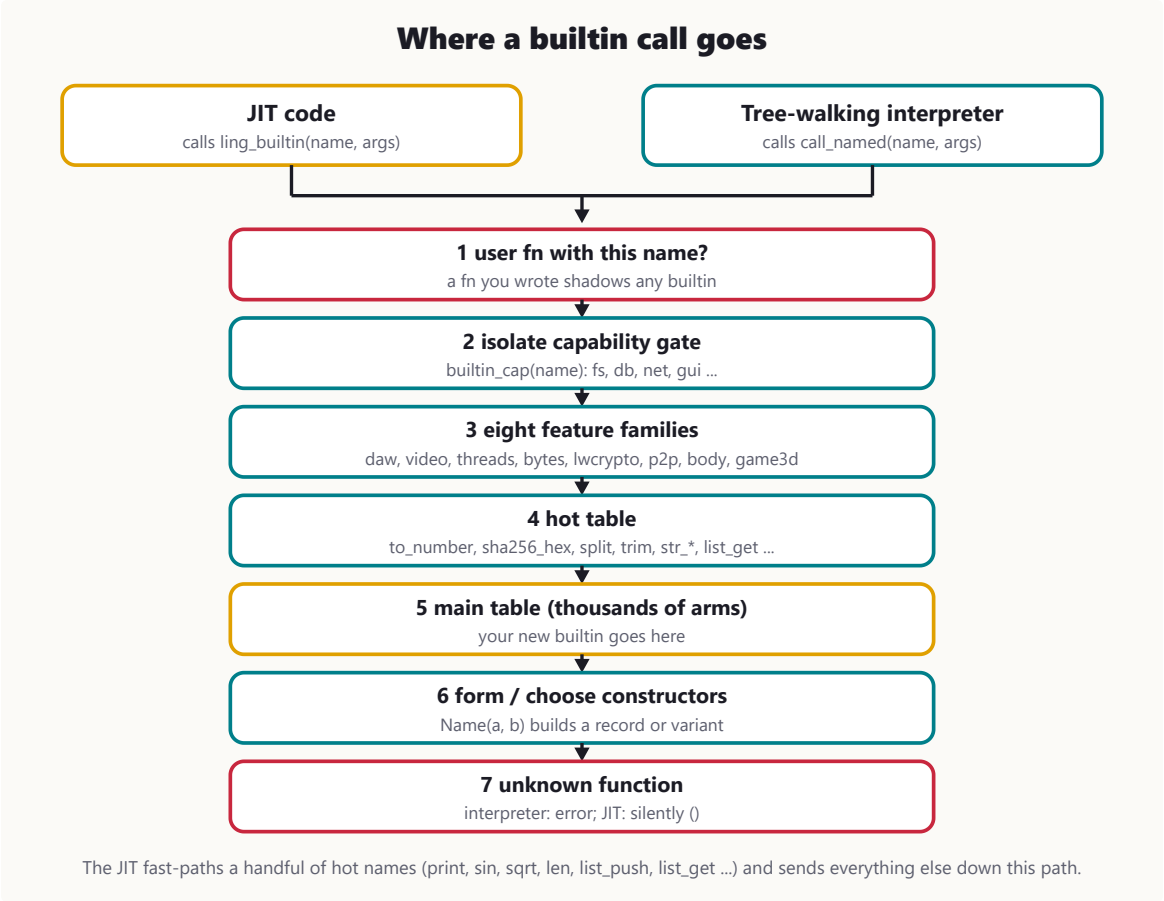
You want to add...	Edit this	Live?
a builtin function	`src/runtime/mod.rs` (or a family file in `src/runtime/`)	yes
a keyword spelling in some human language	`classify_word` in `src/lexer/mod.rs`	yes
an entry-point name (`start` in your language)	`ENTRY_NAMES` in `src/entry.rs`	yes
a builtin alias in your language	extra `|` alternatives on the builtin's match arm	yes

You want to add…	Edit this	Live?
a target for `lingfu normalize`	the `Lang` table in `crates/ling-fu/src/normalize.rs`	yes, separate from the lexer
editor completion, hover and lint data	`editors/vscode-ling/src/lingdata.js`	yes, hand-maintained
a derive macro	`crates/ling-macro`	placeholder
a different lexer or AST	`crates/ling-lex`, `crates/ling-ast`, `crates/ling-polyglot`	not called by the `ling` binary
new vocabulary files	`lexicons/*.ling`	documentation only, never read by the compiler

The last row matters. The repository rules file says the lexicon directory is "the master record" of every keyword, but the compiler never opens those files: spellings come from `classify_word`, builtin aliases from match arms. If you add a language only to `lexicons/xx.ling`, nothing happens. Keep the files in sync for the humans, and change the Rust for the machine.

How a builtin call is dispatched

Everything that is not syntax is a builtin: `print`, `sqrt`, `list_push`, `sha256_hex`, `daw_play`. There is exactly one implementation of every builtin, in the interpreter. The JIT does not duplicate them. Compiled code calls a C-ABI function, `ling_builtin(name, args)`, which decodes the arguments into interpreter values and calls the same `call_named` that the tree-walker uses.



A call is resolved by name through seven stages. A new builtin normally lives in stage 5.

The stages are worth knowing in order, because each one has a consequence:

1. **A user function wins.** If your program defines `fn clamp`, it shadows the builtin `clamp`, on both backends.
2. **The isolate gate.** Code running inside an isolate (see *Concurrency and Isolates*) may not call builtins that need a capability. `builtin_cap(name)` in `src/runtime/threads.rs` maps names to capabilities by prefix: `file_`, `fs_` and `tar_` need `fs`; `db_` needs `db`; `net_`, `http_`, `lingtp_` and `ssh_` need `net`; `key_`, `mouse_`, `px_` and `window_` are main-thread only. A new builtin that touches the disk should carry one of those prefixes or it will escape the sandbox.
3. **Feature families.** Eight functions (`daw_builtin`, `video_builtin`, `threads_builtin`, `bytes_builtin`, `lwcrypto_builtin`, `p2p_builtin`, `body_builtin`, `game3d_builtin`) each return `Option`: `None` means "not mine". The interpreter remembers which family

answered each name, so later calls skip the search.

4. **The hot table.** A short `match` of builtins a frame calls hundreds of times (`to_number`, `split`, `trim`, `sha256_hex`, string slicing, list access). The source comment says a late arm in the main table once cost about 3 microseconds per call, which is why this table exists.
5. **The main table.** A very large `match name { ... }` with thousands of alternatives, including the aliases in a dozen scripts.
6. **Constructors.** `form` and `choose` constructors resolve here, after every builtin. A `form` named like a builtin is therefore never reached.
7. **Unknown.** The interpreter returns an error, `unknown function 'name'`. The JIT converts any error from `ling_builtin` into `()`, which is why typos are silent by default.

You can watch stage 1 with a short program. It defines `clamp` and `sqrt` itself, and calls `max`, which it does not:

```
fn clamp(x, lo, hi) {
  println("my clamp called")
  if x < lo { return lo }
  if x > hi { return hi }
  return x
}

fn sqrt(x) { 0 - 1 }

bind start = do {
  println(clamp(15, 0, 10))
  println(max(3, 9))
  println(sqrt(16))
}
```

```
OUTPUT
my clamp called
10
9
-1
```

Both backends print the same. Shadowing `sqrt` is interesting because the JIT has a native fast path for `sqrt`; the user function still wins because both backends resolve a defined function first (a comment in `call_named_inner` says so). The JIT fast paths themselves are a

short list in `crates/ling-codegen/src/cranelift/translate.rs` (`emit_builtin_call`): `print`, `sin`, `cos`, `sqrt`, `abs`, `floor`, `ceil`, `round`, `time_now`, `len`, and the list operations `list_new`, `list_push`, `list_get`, `list_set`. Everything else takes the generic path. You almost never need to touch that file: a new builtin is correct the moment the interpreter knows it, and the JIT reaches it through `ling_builtin` without any change.

The family-and-route idea is easy to model in a page of Rust. This is not Ling's code; it is a compiled miniature of the dispatch described above, with the same "ask each family, remember the winner" structure:

```
use std::collections::HashMap;

#[derive(Debug, Clone, PartialEq)]
enum Value {
    Number(f64),
    Text(String),
    Unit,
}

type Family = fn(&str, &[Value]) -> Option<Value>;

fn math_family(name: &str, args: &[Value]) -> Option<Value> {
    match (name, args) {
        ("wrap", [Value::Number(x), Value::Number(lo), Value::Number(hi)]) => {
            Some(Value::Number(lo + (x - lo).rem_euclid(hi - lo)))
        }
        _ => None,
    }
}

fn text_family(name: &str, args: &[Value]) -> Option<Value> {
    match (name, args) {
        ("shout", [Value::Text(s)]) => Some(Value::Text(s.to_uppercase())),
        _ => None,
    }
}

const FAMILIES: [Family; 2] = [math_family, text_family];
const MAIN: u8 = FAMILIES.len() as u8;

struct Interp {
    route: HashMap<String, u8>,
    walks: u32,
}

impl Interp {
    fn call(&mut self, name: &str, args: &[Value]) -> Value {
        match self.route.get(name).copied() {
```

```
            Some(MAIN) => {}
            Some(k) => {
                if let Some(v) = FAMILIES[k as usize](name, args) {
                    return v;
                }
                self.route.remove(name);
            }
            None => {}
        }
        if !self.route.contains_key(name) {
            self.walks += 1;
            for (k, f) in FAMILIES.iter().enumerate() {
                if let Some(v) = f(name, args) {
                    self.route.insert(name.to_string(), k as u8);
                    return v;
                }
            }
            self.route.insert(name.to_string(), MAIN);
        }
        Value::Unit
    }
}

fn main() {
    let mut it = Interp { route: HashMap::new(), walks: 0 };
    let n = |x: f64| Value::Number(x);
    println!("{:?}", it.call("wrap", &[n(370.0), n(0.0), n(360.0)]));
    println!("{:?}", it.call("wrap", &[n(-10.0), n(0.0), n(360.0)]));
    println!("{:?}", it.call("shout", &[Value::Text("hi".into())]));
    println!("{:?}", it.call("nope", &[]));
    println!("{:?}", it.call("nope", &[]));
    println!("table walks: {}, remembered routes: {}", it.walks, it.route.len());
}
```

```
OUTPUT
Number(10.0)
Number(350.0)
Text("HI")
Unit
Unit
table walks: 3, remembered routes: 3
```

Five calls but only three table walks: the second `wrap` and the second `nope` were answered from the route cache. The real interpreter does the same, including for names nobody handles.

> **NOTE**
>
> The in-kernel runtime described in *Ling OS and Embedded* has its own `ling_builtin` with the same shape: a per-call-site cache, a core table, and eight registrable extension slots (`lingrt::register`). A builtin you add to `src/runtime/mod.rs` exists on the desktop only; the kernel needs its own implementation.

Adding a builtin: `wrap`

Games and dials constantly need to fold a value into a range: an angle into 0 to 360, a scroll offset into the content length. Ling has `clamp` but no wraparound, so we will add `wrap(x, lo, hi)`, returning a value in the half-open range from `lo` to `hi`.

The patch is one match arm. It goes in the main table next to the other math builtins, just before the arm that handles `pi` and `tau`:

```
+            // wrap(x, lo, hi): fold x into the half-open range [lo, hi), the way an
angle wraps at 360.
+            "wrap" | "wrap_range" => {
+                let x = self.arg_num(&args, 0, 0.0)?;
+                let lo = self.arg_num(&args, 1, 0.0)?;
+                let hi = self.arg_num(&args, 2, 1.0)?;
+                let span = hi - lo;
+                if span <= 0.0 {
+                    return Ok(Value::Number(lo));
+                }
+                return Ok(Value::Number(lo + (x - lo).rem_euclid(span)));
+            },
```

Four things to notice. `arg_num(&args, index, default)` reads a numeric argument and supplies a default when the caller passed fewer; `arg_str` does the same for text. The arm returns early with `return Ok(Value::Number(..))`. The `|` adds an alias, which is exactly how the existing `clamp` arm carries its Chinese, Thai and Hebrew names. And the degenerate case (`hi <= lo`) returns `lo` instead of dividing by zero, because a panic in a builtin closes the program; the neighbouring `clamp` arm has a comment about exactly that.

Building and testing it

Do not experiment inside your only checkout. The recipe that worked here was to copy `src/`, `build.rs`, `Cargo.toml`, `Cargo.lock` and the small `stt/` folder to a scratch directory, make the `crates/...` path dependencies absolute, empty the `[workspace]` member list, and build with a private target directory so nothing collides with your installed toolchain:

```
OUTPUT
$ CARGO_TARGET_DIR=scratch/tgt cargo check --offline --lib      # about 15 s once
dependencies are checked
$ CARGO_TARGET_DIR=scratch/tgt cargo build --offline --bin ling # 6 min 40 s the first
time (dev profile)
```

An incremental rebuild after a change to `src/` took under two minutes. The test program is plain Ling, so we can only show it as a module: it is complete, but it needs the patched binary and fails on a stock `ling`.

```
FRAGMENT
bind start = do {
  print(wrap(370, 0, 360))
  print(wrap(0 - 10, 0, 360))
  print(wrap(5.5, 1, 4))
  print(wrap_range(7, 0, 5))
  print(wrap(3, 5, 5))
}
```

Run with the patched build and with the stock `ling` 2030.1.53 on the same file:

```
OUTPUT
patched, JIT          patched, --interp       stock, JIT     stock, --interp
10                    10                      ()             error[runtime]: unknown
function 'wrap'
350                   350                     ()
2.5                   2.5                     ()
2                     2                       ()
5                     5                       ()
```

Look at the stock JIT column. Nothing failed; five lines of `()` were printed and the program exited with status 0. This is the single most important property to understand when you extend Ling: **forgetting to register something never produces an error on the default backend**. Always run a new builtin under `--interp` as well, and write the check so that a `()` is visibly wrong.

> TIP
>
> Name new builtins with the prefix that gives them the right isolate capability (`fs_`, `net_`, `db_`) from day one. Renaming a public builtin later breaks every script that uses it, and the capability table matches prefixes, not intentions.

Rules of thumb for builtins

- **Soft failure.** Builtins in this codebase return a neutral value on bad input (`to_number("x")` is `0`, a missing file gives `()` or `false`) instead of raising. Raise an `Err(EvalErr)` only for programmer errors you want to surface on `--interp`.

- **Feature gating.** A builtin that pulls in a heavy dependency is wrapped in `#[cfg(feature = "...")]`, the way the `web` feature gates SQLite and `run_process`. Without the feature the name falls through to stage 7, and users see the silent-`()` behaviour. Document the feature in the builtin's chapter.
- **Families.** If you are adding a dozen related builtins with a shared prefix, make a family: a new file `src/runtime/yours.rs` with `pub(super) fn yours_builtin(&mut self, name: &str, args: &[Value]) -> Option<EvalResult>` that starts with a prefix test and returns `None` otherwise, then add it to the `FAMILIES` array (and bump its length). `game3d.rs` is the model: its guard is `name.starts_with("tm_") || name.starts_with("coll_") || ...`, so the cost for every other name is one failed prefix test, paid once thanks to the route cache.
- **Order matters.** In a Rust `match`, the first arm that matches wins and a duplicate name later is unreachable. The compiler warns, and the warning is your only notice, so keep the build warning-free.
- **Do the paperwork.** Add the name to `lexicons/en.ling` (for humans), to `editors/vscode-ling/src/lingdata.js` (for completion and lint, or your own function will be flagged as unknown, as shown below), to the reference chapter, and write a test.

Adding a language to the polyglot lexicon

Ling's "polyglot" feature is smaller and more concrete than it sounds. There is no translation layer. The lexer, `src/lexer/mod.rs`, reads a word and calls `classify_word`, one giant `match` from spelling to token. English `bind`, Chinese `令`, Japanese `束縛`, Thai `ผก`, Russian `связать` and Arabic `ربط` all map to `Token::Bind`, in the same file. Anything not listed becomes `Token::Ident`. The parser sees only tokens and never knows what language produced them, so one file can mix languages freely.

To add Latin, then, we add arms. Because the target tokens already exist, the whole language is a dozen lines, plus one line in `src/entry.rs` so that `initium` counts as an entry name, and a `"scribe"` alias on the `print` arm:

```
+            // Latin (a teaching example: every spelling resolves to an existing
token)
+            "ligare" => Token::Bind,
+            "facere" => Token::Do,
+            "functio" => Token::Fn,
+            "quodsi" => Token::If,
+            "ceterum" => Token::Else,
+            "dum" => Token::While,
+            "redde" => Token::Return,
+            "verum" => Token::Bool(true),
+            "falsum" => Token::Bool(false),
             // Everything else is an identifier
             other => Token::Ident(other.to_string()),
```

```
     "התחל", // Hebrew "start"
+    "initium", // Latin "start"
```

```
-            | "печать" => {
+            | "печать" | "scribe" => {
```

This was built into the same scratch copy (a second build, under two minutes). The program below is complete Latin source; on the patched binary it runs on both backends, and on the stock binary it stops at the first word:

FRAGMENT

```
functio quadratum(n) {
  redde n * n
}

ligare initium = facere {
  ligare i = 1
  dum i <= 3 {
    quodsi i == 2 {
      scribe("secundus: ", quadratum(i))
    } ceterum {
      scribe("numerus ", i, " ", verum)
    }
    ligare i = i + 1
  }
}
```

OUTPUT

```
patched, JIT and --interp        stock ling
numerus 1 true                   error[parse]: unexpected token at top level:
Ident("functio")
secundus: 4
numerus 3 true
```

Note that `scribe(..)` printed correctly even though the JIT's fast-path list contains only `print` and a handful of translations. Names not on that list go through `ling_builtin`, reach the interpreter's `print` arm, and print identically. Aliases never need JIT work.

Both runs printed a line `[detected language: Portuguese (Português)]` on standard error. It comes from `ling::detect_language`, a heuristic that looks at the source text before running, and is informational only. It guessed Portuguese because `ligare` resembles `ligar`. If you add a language seriously, add it to the detector as well, or users will see the wrong banner.

The costs of a new keyword

- **Every spelling becomes reserved for every program.** `si` is Spanish for `if`, so no Ling program can have a variable called `si`; `para`, `form`, `can`, `fit` and `give` are in the same position. The LingStationX developer notes list "reserved words in every supported

language" as the first language gotcha they hit, and the symptoms (`expected LBrace, got Ident`) do not mention the real cause. Pick spellings that are unlikely identifiers, or accept the cost.

- **`lingfu normalize` has its own table.** `lingfu normalize <lang>` rewrites a whole project (contents and file names) into one language, but it uses the `Lang` enum and word tables in `crates/ling-fu/src/normalize.rs`, which knows twelve languages. A new lexer spelling does not appear there until you add it.
- **Fixing it later is expensive.** Changing a spelling in a released lexicon breaks every program that used it. Treat additions as permanent.

A custom lexicon without recompiling

Patching the compiler is the right answer for a language the project wants to ship. For a team convention, a teaching dialect or an in-house vocabulary, you can get most of the effect without touching Rust: write a source-to-source translator in Ling, run it as a build step, and feed the result to `ling run`.

The program below is the whole tool. It carries its own lexicon text and its own Latin source as strings so that it is self-contained. It scans the source character by character, collecting identifier-like words, replaces whole words found in the lexicon, and leaves string literals untouched, so the `dum` inside a string survives. It then writes the translated program to disk and runs it with `run_process`, capturing the child's output in a log.

```
# A user-level lexicon: translate a Latin-keyword program into plain Ling, then run it.
bind LEXICON = "ligare=bind
facere=do
functio=fn
quodsi=if
ceterum=else
dum=while
redde=return
scribe=print
initium=start"

bind SOURCE = "functio quadratum(n) {
  redde n * n
}

ligare initium = facere {
  ligare i = 1
  dum i <= 3 {
    quodsi i == 2 {
      scribe(\"secundus: \", quadratum(i))
    } ceterum {
      scribe(\"dum stays inside this string \", i)
    }
    ligare i = i + 1
  }
}"

bind WORD_CHARS = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_"

fn is_word_char(c) { str_find(WORD_CHARS, c) >= 0 }

fn load_lexicon(text) {
  bind names = []
  bind words = []
  bind lines = str_split(text, "\n")
  bind i = 0
  while i < len(lines) {
    bind eq = str_find(lines[i], "=")
```

```
    if eq > 0 {
      bind names = list_push(names, str_slice(lines[i], 0, eq))
      bind words = list_push(words, str_trim(str_slice(lines[i], eq + 1, len(lines[i])
- eq - 1)))
    }
    bind i = i + 1
  }
  bind table = [names, words]
  table
}

fn lookup(word, lex) {
  bind i = 0
  while i < len(lex[0]) {
    if lex[0][i] == word { return lex[1][i] }
    bind i = i + 1
  }
  return word
}

fn translate(src, lex) {
  bind out = ""
  bind word = ""
  bind in_str = false
  bind i = 0
  while i < len(src) {
    bind c = str_slice(src, i, 1)
    if in_str {
      bind out = out + c
      if c == "\"" { bind in_str = false }
    } else if c == "\"" {
      bind out = out + lookup(word, lex) + c
      bind word = ""
      bind in_str = true
    } else if is_word_char(c) {
      bind word = word + c
    } else {
      bind out = out + lookup(word, lex) + c
      bind word = ""
```

```
    }
    bind i = i + 1
  }
  out + lookup(word, lex)
}

bind start = do {
  bind plain = translate(SOURCE, load_lexicon(LEXICON))
  println(plain)
  println("-----")
  write_file("out/latin_plain.ling", plain)
  run_process("ling", ["run", "out/latin_plain.ling"], 30, "out/run.log")
  print(str_trim(read_file("out/run.log")))
}
```

OUTPUT

```
fn quadratum(n) {
  return n * n
}

bind start = do {
  bind i = 1
  while i <= 3 {
    if i == 2 {
      print("secundus: ", quadratum(i))
    } else {
      print("dum stays inside this string ", i)
    }
    bind i = i + 1
  }
}
-----
dum stays inside this string 1
secundus: 4
dum stays inside this string 3
```

The output is identical on `--interp`. (The `[detected language: ...]` banner goes to standard error and is not part of the output above.) Three details earned their place by failing first. `str_slice(s, start, length)` takes a length, not an end index. A function body whose last line starts with `[` continues the previous expression, so `[names, words]` after a `while` loop was a parse error (`expected RBracket, got Comma`); binding the pair to `table` first fixes it. And `run_process` returns the child's exit code, so a real tool would check for `0` before trusting the log.

> **WARNING**
>
> A word-level translator knows nothing about scope. If a Latin program names a variable `dum`, the tool turns it into `while`. Reserve every lexicon word, exactly as the compiler does.

`ling-macro` and the other placeholders

Ling programs have no macro system, and the `ling-macro` crate does not change that. Its entire source is one derive macro that parses its input and returns nothing:

```
#[proc_macro_derive(LingType)]
pub fn derive_ling_type(input: TokenStream) -> TokenStream {
    let _input = parse_macro_input!(input as DeriveInput);
    // Generate Ling type metadata
    TokenStream::new()
}
```

`#[derive(LingType)]` compiles and generates nothing. A search of the source tree found no use of `ling_macro` or `LingType` outside the crate itself, so neither the `ling` binary nor any other crate calls it. The same is true of `ling-core` (arena, error and type definitions for a type checker that `ling run` never invokes), `ling-lex`, `ling-ast` and `ling-polyglot`, which are parallel, smaller front ends. The root crate's own `src/semantic` and `src/borrowck` are likewise not on the `run`, `build` or `lingc` path, which is why `own`, `lend` and type annotations are accepted and ignored. If you need compile-time checks today, build them as a separate tool that parses with the `ling` crate (see *Using the Crates from Rust*), not as a patch to these.

This is useful negative knowledge: the productive extension points are `src/lexer`, `src/runtime` and `crates/ling-fu`. Everything else is scaffolding for the future.

Editor tooling: the VS Code extension

`editors/vscode-ling` is a small, readable JavaScript extension (about 800 lines in total) and a good first contribution target. Its parts:

- `syntaxes/ling.tmLanguage.json`: the TextMate grammar (comments, strings, numbers, keywords in many scripts, builtins).
- `snippets/ling.json`: ten snippets (`start`, `if`, `while`, `match`…).
- `src/lingdata.js`: pure data. Keywords, types, constants, builtins grouped by category (388 names in total) and a table of signatures used for hover, signature help and inlay hints.
- `src/extension.js`: completion, hover, signature help, document symbols, inlay parameter-name hints, and a **linter**: unbalanced brackets, `let` (which Ling does not have), and calls to unknown functions.
- `test/lint.test.js`: an offline test that stubs the `vscode` module, so you can run `node test/lint.test.js` without an editor. It reported `All 15 checks passed` on a scratch copy.

The unknown-call lint is exactly the tool Ling needs, because on the JIT a typo is invisible at run time. We can drive the linter directly with the same stub approach. The harness below loads the extension, feeds it a file and prints diagnostics:

```
const Module = require("module");
class Position { constructor(o) { this.off = o; } }
class Range { constructor(s, e) { this.start = s; this.end = e; } }
class Diagnostic { constructor(r, m, s) { this.range = r; this.message = m;
this.severity = s; } }
const stub = { DiagnosticSeverity: { Error: 0, Warning: 1 }, InlayHintKind: {
Parameter: 2 },
  Position, Range, Diagnostic, InlayHint: class {},
  workspace: { getConfiguration: () => ({ get: (_k, d) => d }) } };
const orig = Module._load;
Module._load = (r, p, m) => (r === "vscode" ? stub : orig.call(Module, r, p, m));
const { _internal } = require("./vscode-ling/src/extension.js");
const text = require("fs").readFileSync(process.argv[2], "utf8");
const doc = { languageId: "ling", getText: () => text, positionAt: o => new
Position(o), offsetAt: p => p.off };
const lineOf = off => text.slice(0, off).split("\n").length;
for (const d of _internal.lint(doc, { unknownCalls: true }))
  console.log(`line ${lineOf(d.range.start.off)}: ${d.severity === 0 ? "error" :
"warning"}: ${d.message}`);
```

Fed a program with two typos and a nested one:

```
FRAGMENT
fn square(x) { x * x }

bind start = do {
  bind n = 7
  bind a = squar(n)
  bind b = square(n)
  bind c = tm_foo(1)
  bind d = draw_rect(1 2 3 4)
  println(b)
  print(squar(n))
}
```

the linter prints:

```
OUTPUT
line 5: warning: Unknown function `squar` — not a builtin or a defined binding.
line 7: warning: Unknown function `tm_foo` — not a builtin or a defined binding.
line 9: warning: Unknown function `println` — not a builtin or a defined binding.
```

Reading the report like a reviewer gives four findings, all reproducible, and all excellent first pull requests:

1. `println` **is flagged unknown**, although it is a real builtin (`print` and `println` share one runtime arm). The data tables list `print` only.
2. `squar` **on line 10 is not reported.** The call regex consumes the character before the name, so `print(squar(n))` matches `print(` and the `(` is gone before `squar` is tried. Any call nested in another call, or directly after a `(`, is invisible to the linter. Replacing the pattern with a look-behind, `/()(?<![.\p{L}\p{N}_])([a-z][a-z0-9_]*)\s*\(/gu`, makes it report nested calls, and the extension's own 15 checks still pass. (After the fix it also reports `sha256_hex` and `str_to_hex`, which are real builtins missing from the data table, a second symptom of finding 1.)
3. `draw_rect` **is not flagged, yet it does not exist.** `lingdata.js` lists `draw_rect`, `draw_text` and `panic` among the known builtins. Run on `--interp`, the program reports `unknown function 'draw_rect'`, and on the JIT it silently yields `()`. The editor vouches for names

the runtime lacks. A scan that checks each editor name for a string literal in `src/runtime/*.rs` flags 108 candidates; some are shape builtins registered through other tables, so treat the list as a work queue, not a verdict.

4. **The snippet table is wrong in places.** The `Function` snippet expands to `bind name = (n: number) -> number { ... }`, which fails to parse (`expected RParen, got Colon`), and the keyword descriptions say `stop` breaks a loop and `again` continues. Both keywords are inert in the current parser: a `while` loop containing `if i == 3 { stop }` printed `1 2 3 4 5`.

The cure for the whole class is to generate `lingdata.js` from the runtime instead of maintaining it by hand: extract every string literal on a match arm, subtract names known to be aliases, and publish. Until someone does that, the practical rule is to treat editor warnings as hints and the interpreter's `unknown function` as the truth.

> **TRY IT**
>
> Apply the look-behind fix in your own copy of `src/extension.js`, add `println`, `sha256_hex` and `str_to_hex` to the `I/O` and `Text` categories in `lingdata.js`, add a test case for the nested call to `test/lint.test.js`, and confirm `node test/lint.test.js` still reports all checks passing. Packaging is `vsce package` (the repository ships two prebuilt `.vsix` files); that step was not run for this book.

The Python bindings: `ling-py`

`crates/ling-py` is a PyO3 module (`crate-type = ["cdylib"]`, `pyo3 = 0.22`) that exposes three classes and three functions to Python:

Name	Wraps	Purpose
`AudioEngine`	`ling_audio::AudioEngine`	4D spatial tones: `set_tone`, `clear_tone`, listener orientation
`HyperbolicWorld`	`ling_physics::hyperbolic`	gravity direction in a Poincare-ball world
`hyp_distance`, `exp_map`, `log_map`	`ling_physics::hyperbolic`	stand-alone hyperbolic geometry
`NetClient`	`tokio-tungstenite` directly	WebSocket netplay client with a background thread, `connect`, `send`, `try_recv`

Two corrections to the README. It says the crate wraps `ling-net`; it does not, since `Cargo.toml` depends on `ling-audio`, `ling-physics`, `tokio` and `tokio-tungstenite` (and `ling-net` is absent). And `NetClient` speaks WebSocket text frames only: it does not talk lingtp or

any of `ling-net`'s other transports. The module itself is three lines of registration:

```
#[pymodule]
fn ling_py(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_class::<PyAudioEngine>()?;
    m.add_class::<PyHyperbolicWorld>()?;
    m.add_class::<PyNetClient>()?;
    m.add_function(wrap_pyfunction!(hyp_distance, m)?)?;
    m.add_function(wrap_pyfunction!(exp_map, m)?)?;
    m.add_function(wrap_pyfunction!(log_map, m)?)?;
    Ok(())
}
```

`cargo check` on a scratch copy of the crate (with the two path dependencies made absolute) finished cleanly against a Miniconda interpreter. We did not link the extension module or import it from Python, so treat runtime behaviour, especially `NetClient` against a live server, as untested here. To add a class, write a `#[pyclass]` with `#[pymethods]`, then one more `add_class` line.

Contributing

The repository's `rules.md` is short and opinionated; the parts that affect a first patch are zero compiler warnings, tests and benchmarks passing, no commented-out code, documentation updated together with the code (keywords, syntax and builtins), and commits that are concise and human. It also says source files should stay under a thousand lines, a rule the 19,000-line `src/runtime/mod.rs` visibly predates; new code should follow the rule by living in family files. CI lives in `.github/workflows` (`ci.yml`, `docs.yml`). Tests are Rust: 86 `#[test]` functions inside `src/` and two integration files under `tests/`, one of which runs complete `.ling` examples through `ling::run`.

A workable first-contribution loop, all of it done in this chapter:

1. Copy the tree (or fork) and get `cargo check --lib` green with a private `CARGO_TARGET_DIR`.
2. Make the smallest change, one match arm, one lexer line, one data-table entry.
3. Prove it on both backends, with a program that prints something visibly wrong on failure.
4. Update the lexicon file, the editor data and the docs.
5. Keep the diff warning-free, then open a pull request that says what you ran.

Chapter summary

- Builtins are one implementation in the interpreter; the JIT reaches them through `ling_builtin` by name, so a new arm works on both backends with no JIT change.
- Dispatch order: user function, isolate capability gate, eight families, hot table, main table, constructors, then unknown (an error on `--interp`, a silent `()` on the JIT).
- To add a builtin, add a match arm (with aliases) in `src/runtime/mod.rs` or a family file; name it with the right capability prefix; verify with `--interp`.
- A language is arms in `classify_word` plus `ENTRY_NAMES`; `lexicons/*.ling` are never read, `lingfu normalize` has a separate table, and every spelling you add becomes a reserved word.
- A user-level lexicon, a translator written in Ling and run with `run_process`, covers conventions and dialects without recompiling.
- `ling-macro`, `ling-core`, `ling-lex` and friends are not on the execution path; patch `src/lexer`, `src/runtime` and `crates/ling-fu`.
- The VS Code extension is a good place to contribute: fix the nested-call lint, add missing names, and remove names the runtime does not have.
- `ling-py` wraps `ling-audio`, `ling-physics` and a WebSocket client, not `ling-net`; it type-checks but was not imported in Python for this book.

Exercises

1. Add a builtin `lerp_wrap(a, b, t)` that interpolates `a` to `b` along the shorter way around a circle of 360 degrees. Where do you put it, which two programs prove it, and what must you not forget for the editor?
2. Add the Italian keyword spelling `mentre` as another way to write `while`. List every file you would touch for it to work with `ling run`, and every extra file so that `lingfu normalize` could also target it.
3. A teammate says "I added my new builtin and `ling run` prints `()` for it, so the arm must be wrong." Give three different causes, and the one-command check that separates them.

Answers

1. Put the arm in the main table beside `wrap` (or in a math family file), reading three arguments with `arg_num`; compute `d = (b - a).rem_euclid(360.0)`, subtract 360 when `d > 180.0`, and return `a + d * t` (wrapped). Prove it with one program on `ling run` and the same program on `ling run --interp`, using values whose answer crosses 0/360 (for

example `lerp_wrap(350, 10, 0.5)` should be `0` or `360`, never `180`). For the editor, add the name to `lingdata.js` (category and signature) so the linter does not flag it, and to the reference and `lexicons/en.ling`.

2. For `ling run`: one arm line, `"mentre" => Token::While`, in `classify_word` in `src/lexer/mod.rs` (check no earlier arm uses the word). Extra for `lingfu normalize`: `Lang` has no Italian entry, so add the variant and its word tables in `crates/ling-fu/src/normalize.rs`; optionally the language detector, and a `lexicons/it.ling` file for documentation. Remember `mentre` is now unusable as an identifier everywhere.
3. (a) The binary being run is the stock one, not the one you built: check which executable `ling` resolves to. (b) The arm is below an earlier arm, family or hot-table entry that returns first, or is behind a `#[cfg(feature = ...)]` the build did not enable. (c) The JIT is hiding an error: the interpreter would say `unknown function` or a real runtime error. The one-command check is `ling run --interp yourfile.ling`; it prints the message the JIT swallowed.

CHAPTER 10

Ling OS and Embedded

What you'll learn

Ling is not only a scripting language that happens to have a window. The same source tree contains a kernel, a freestanding language-model engine, a tracker synthesizer that runs without an operating system, and desktop applications that are compiled into a bootable image. This chapter explains what is in that stack, what runs where, and how you write an app once and ship it on a desktop and inside the kernel.

It is also a chapter about honesty. Booting an operating system was not part of what was done for this book, so every claim here is labelled by how it was established: **run** (we executed it), **compiled** (we built it but did not exercise it), or **documented** (the repository says so and we did not check). Where a feature is described in a README as complete and its own source comments say "not here yet", we tell you both.

By the end you will be able to read a LingOS kernel manifest, split an application into a shared body and a small platform layer, run the freestanding engines on your own machine and in `no_std` mode, and judge for yourself how much of LingOS to trust for a given project.

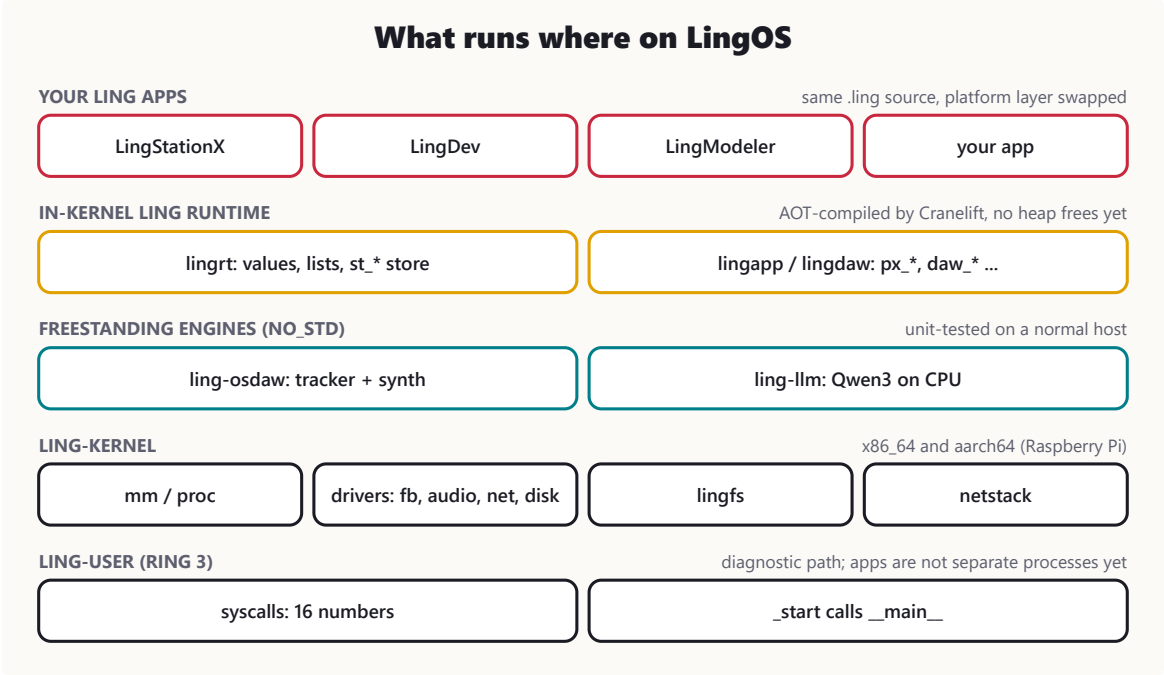
What LingOS is

LingOS is a from-scratch operating system whose desktop and applications are Ling programs, running on a Rust kernel. Its own README describes it as booting on x86_64 (BIOS and GRUB, QEMU, VirtualBox) and on Raspberry Pi (aarch64, QEMU `raspi3b` and real hardware), with these parts: interrupts and a cooperative scheduler, a ring-3 layer with 16 system calls, a buddy allocator and 4-level paging, a window manager with a dock and a per-app audio mixer, an e1000 network driver with a minimal TCP and HTTP client, and `lingfs`, a content-addressed root filesystem keyed by BLAKE3 hashes.

The repository is unusually frank about its limits, and you should take its caveats at face value:

- "`apps/*` remain `use`-shared source compiled into kernel targets, not separate processes yet." An app is part of the kernel image, not a program the kernel loads.

- Packages are an unsigned, length-prefixed blob format over plain HTTP with no dependency resolution; "no execution".
- The Raspberry Pi 5 needs a hardware-abstraction follow-up: LingDev's README says LingOS's aarch64 HAL targets the Pi 3 and 4 memory map today.



The LingOS stack. Everything above the kernel is `.ling` source or a freestanding Rust crate.

The stack, top to bottom:

Layer	What it is	Status
Ling apps	LingStationX, LingDev, LingModeler, an installer, a shell, a messenger	documented
In-kernel Ling runtime	`lingrt` (values, lists, `st_*` store), `lingapp` (window, `px_*` drawing, fonts, input), `lingdaw` (the `daw_*` project model)	compiled
Freestanding engines	`ling-osdaw` (tracker and synth), `ling-llm` (Qwen3 inference)	run and compiled
`ling-kernel`	memory, processes, drivers, `lingfs`, network stack, an app runtime of about 3,600 lines	documented
`ling-user`	ring-3 runtime: syscalls and `_start`	compiled

The freestanding crates

The crates you can touch without a kernel are the most useful part of this chapter, because they show what "freestanding" means in practice: `#![no_std]`, a heap through `alloc`, maths through `libm`, and nothing that needs an operating system.

`ling-osdaw`: a synthesizer with no OS underneath

`ling-osdaw` is the music engine behind LingOS's built-in LingStation window: 8 channels by 64 rows, 14 instruments (saw lead, bass, pluck, pad, FM bell, kick, snare, hat, supersaw, growl, voice, organ, clap, tom), a master effect rack, note scripts, MIDI import and export, and a WAV renderer, all in `no_std` plus `alloc`. Its crate documentation is explicit about what the desktop LingStationX has that it does not: a language runtime, audio input, threads, a file picker, TTF text, speech and language-model tools, sample import, and the large preset library.

Because it needs no OS, it also runs on a normal host, which is how its authors test it. We copied the crate to a scratch directory and ran its own tests:

```
OUTPUT
$ cargo test --offline
running 7 tests
test result: ok. 7 passed; 0 failed; ...
running 7 tests
test result: ok. 7 passed; 0 failed; ...
```

Fourteen tests passed. A host program is a better demonstration of what the crate is, so here is one that loads the built-in demo song, serializes it, reloads it, renders it to a WAV image in memory, and inspects the result:

```
use ling_osdaw::{Engine, Song, INSTRUMENTS, SAMPLE_RATE};

fn main() {
    let song = Song::demo();
    println!("bpm {} rows {} instruments {}", song.bpm, song.length,
INSTRUMENTS.len());
    println!("length {:.2} s (+{:.2} s tail) at {} Hz", song.seconds(),
song.tail_seconds(), SAMPLE_RATE);

    let bytes = song.to_bytes();
    let back = Song::from_bytes(&bytes).expect("round trip");
    println!("serialized {} bytes, reload bpm {}", bytes.len(), back.bpm);

    let wav = Engine::render_wav(&song);
    let peak = wav[44..]
        .chunks_exact(2)
        .map(|b| i16::from_le_bytes([b[0], b[1]]).unsigned_abs())
        .max()
        .unwrap_or(0);
    println!("{} bytes of {}, peak {}", wav.len(), String::from_utf8_lossy(&wav[0..4]),
peak);
}
```

```
OUTPUT
bpm 122 rows 32 instruments 14
length 3.93 s (+5.50 s tail) at 48000 Hz
serialized 1068 bytes, reload bpm 122
1811452 bytes of RIFF, peak 24482
```

A whole song is 1,068 bytes. The renderer produced 1.8 MB of 16-bit stereo audio from it with no file, thread or device involved. That is the design: the kernel's mixer calls `Engine::render` for one audio block at a time, and everything musical lives in the crate where it can be unit-tested on a laptop. (The program above lives in a Cargo project whose `ling-osdaw` dependency is a path dependency on the workspace copy; if you use the published crate, depend on it by version.)

The same crate also type-checks for the bare-metal target: `cargo check --target x86_64-unknown-none` finished cleanly in about two seconds. That is the second half of the freestanding promise: not only does it avoid `std`, the compiler proves it.

`ling-llm`: a language model with no OS

`ling-llm` is "a freestanding Qwen3 inference engine": `no_std` plus `alloc` plus `libm`, no GPU. It reads the model, its tokenizer and its vocabulary out of one GGUF file, supports F32, F16, Q8_0, Q4_K and Q6_K tensors, builds a byte-level BPE tokenizer from the GGUF's own tokens, runs the transformer with an f16 key-value cache, and offers greedy and temperature/top-p sampling plus the ChatML prompt that `ling-ai` also builds. Weights are held in chunks of at most 8 MiB because the kernel's frame allocator has no larger blocks.

What we established: the crate **compiled** for `x86_64-unknown-none` with `cargo check`, with no features, in a bit over two seconds. What we did not establish: any inference, since there is no model file in this environment. The README's own numbers, which we report without having reproduced them, are a tokenizer identical to the Hugging Face `tokenizers` crate on 14 awkward strings, first-logit cosine similarity of 0.9985 against `candle`, ten identical greedy tokens, and about 16 tokens per second on a desktop CPU with the 0.6B model.

The LingOS README adds the operational reality of using it in the kernel, and these limits are worth reading before you plan anything around it:

- The weights are not in the kernel image. A script (`live/make-model-disk.ling`, run with `ling run --interp`) writes `dist/models.img`, which is attached as a second IDE drive; only the IDE path can read it, not AHCI.
- The disk driver is ATA PIO, so loading 600 MB takes a minute or two.
- Under QEMU's software emulation generation takes "seconds per token"; you need a hypervisor accelerator for real speed.
- In the Terminal, `ask <question>` loads the model on first use and streams the answer.

`ling-user`: ring 3

`ling-user` is the userland runtime for programs that run unprivileged: `#![no_std]`, a heap, strings, collections, a framebuffer helper and the 16 syscalls (`exit`, `write`, `read`, `open`, `close`, `lseek`, `mmap`, `munmap`, `spawn`, `waitpid`, `yield`, `getpid`, `sleep_ms`, `poll_input`, `fb_map`, `uname`). Its entry point is the whole interface between a compiled Ling program and the kernel:

```
#[no_mangle]
pub unsafe extern "C" fn _start() -> ! {
    let ret = __main__();
    let code = if (ret >> 56) == 0x7F {
        0
    } else {
        (f64::from_bits(ret) as i32) as u64
    };
    crate::syscall::ling_sys_exit(code);
    loop {}
}
```

`__main__` is the compiled `bind start = do { ... }`; you met it in *How Ling Runs* in the MIR dump. The return value uses the runtime's NaN-boxed encoding: a number returns as its raw `f64` bits and becomes the exit code, while any tagged value (the top byte `0x7F`, which is unit, a boolean or a heap object) exits with status 0. The crate type-checks on `x86_64-unknown-none`. The repository says ring-3 currently drives only diagnostics (`proctest`, `ps`), and `LingOS/apps/hello` is the example that calls `ling_sys_getpid()` and `ling_sys_exit(0)`.

> **NOTE**
>
> Three things in this chapter compile for `x86_64-unknown-none`: `ling-osdaw`, `ling-llm` and `ling-user`. That is a useful smoke test for any freestanding crate of your own, and it takes seconds. `rustup target add x86_64-unknown-none` is all you need.

How `.ling` runs in the kernel

Two paths exist, and they share one runtime.

Compiled apps. `ling build kernel/x86_64-wm --platform kernel --out dist` compiles the kernel's `main.ling` and the apps it bundles through the Cranelift ahead-of-time backend. The generated code uses the same NaN-boxed value encoding as the host JIT and reaches every builtin through a function named `ling_builtin(name, args)`. In the kernel that function lives in `lingrt.rs`, and it mirrors the host design from *Extending Ling*: a per-call-site cache remembers which table answered a name last time, a core table handles the common names, and `lingrt::register` adds up to eight extension tables (pixel drawing, input, files, the DAW engine). A name nobody answers is reported once and evaluates to unit.

`ling run` in the Terminal. The kernel also contains a Ling front end of its own (`src/lingc`: lexer, parser, `use` flattening, lowering to MIR, all `no_std` copies of the compiler's own sources) and a MIR interpreter over the same runtime. That is how the desktop Terminal's `ling run`, `ling eval` and `ling demo` work, and why the image bakes in sample programs (`fib.ling`, `spiral.ling`, `gfx.ling`).

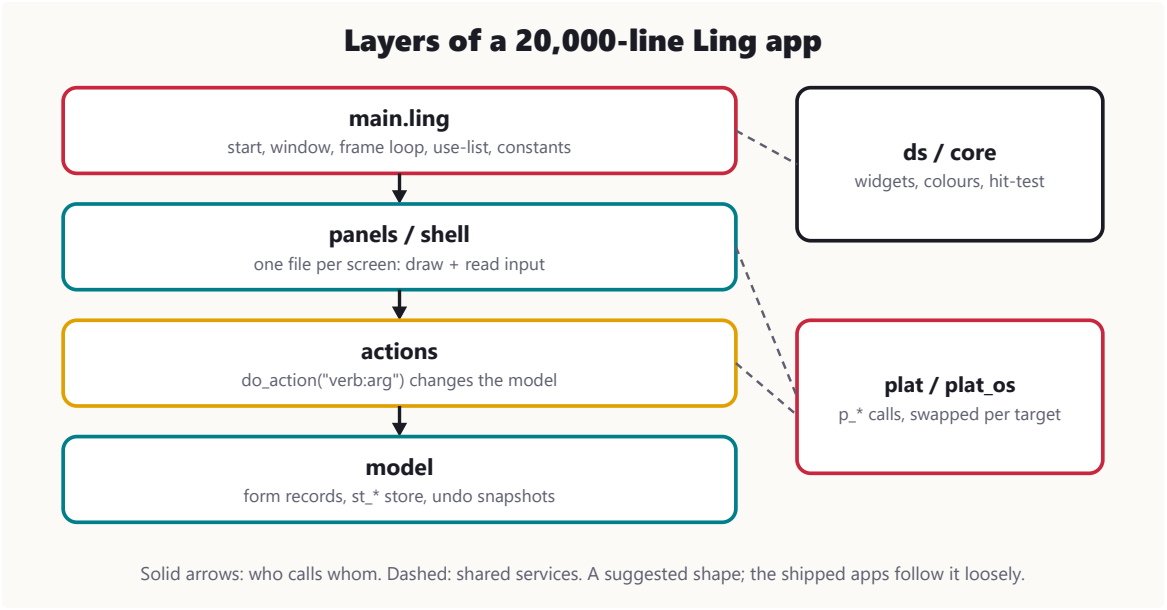
The runtime's header comment states the compatibility contract in one sentence: semantics "follow the host interpreter", not the JIT. Concretely, number display drops `.0`, `len` counts characters, `+` concatenates when either side is text, lists are values (`list_push` returns a new list), and truthiness treats `0`, `""`, `[]` and unit as false.

> **WARNING**
>
> That truthiness rule differs from the default desktop backend. On the JIT, empty text and empty lists are truthy; in the kernel, as on `--interp`, they are falsy. An app that depends on either is not portable. Test with `ling run --interp` on the desktop, which is what the shipped apps do, and write conditions explicitly (`len(xs) > 0`).

And one more sentence from the same comment, which governs how you write kernel code: "**Memory:** nothing is freed yet." The ahead-of-time code emits no drops, so every string and list an app creates stays allocated until the machine is powered off.

One app, two platforms

LingStationX, LingDev and LingModeler each run on the desktop `ling` runtime and inside LingOS. They do it without forking the app, by funnelling everything platform-specific through one small file. LingModeler's README states the whole mechanism in two lines: `ling/main_os.ling` "is generated from `main.ling` by `tools/make_os_entry.py` (it swaps in `plat_os.ling` …)", and the generator is a few lines long: assert that `main.ling` contains `use "plat.ling"`, replace that one line with `use "plat_os.ling"`, prepend a "GENERATED, do not edit" header, and write the result.



The platform layer (bottom right) is a shared service: everything that touches the OS calls `p_*` functions and never asks which OS it is on.

What lives behind the line is instructive, because it is a precise list of what the kernel runtime does not have. In LingModeler's `plat_os.ling` the comments say it directly:

```
FRAGMENT
# plat_os.ling: LingOS platform layer (in-kernel ling runtime: lingrt / lingapp)
# The kernel has px_* drawing, fonts, files and dialogs but no polygon fill, no images
and
# no byte files. Polygons are scan-converted here with px_hline; image features report
"unavailable".

# no list_join in the kernel runtime: join pairwise (halves the list each pass)
fn p_join(l, sep) { ... }
```

So: no polygon fill (the app rasterizes polygons itself, scanline by scanline, with even-odd crossings), no image loading or saving, no byte-level file I/O, no `list_join`. LingDev's README gives the corresponding game-side summary for its LingOS build: "software renderer, no audio, read-only files". Text input also differs: the kernel hands out only what was typed since the last call, so the app keeps its own edit buffer and processes backspace itself.

The technique is worth copying whatever your target. The program below is a runnable miniature: it contains a desktop platform file, a "kernel-style" platform file whose `p_join` is LingModeler's pairwise-halving join, and one shared `main`. It writes both variants to disk, generates the OS entry by replacing exactly one `use` line, and runs each with `run_process`.

```
# Generates and runs both platform variants of one tiny app, the way LingModeler does.
bind PLAT = "fn p_name() { \"desktop\" }
fn p_join(l, sep) { list_join(l, sep) }
"
bind PLAT_OS = "fn p_name() { \"kernel-style\" }
fn p_join(l, sep) {
  if len(l) == 0 { return \"\" }
  bind cur = []
  bind i = 0
  while i < len(l) { bind cur = list_push(cur, \"\" + list_get(l, i)) bind i = i + 1 }
  while len(cur) > 1 {
    bind nxt = []
    bind i = 0
    while i < len(cur) {
      if i + 1 < len(cur) { bind nxt = list_push(nxt, list_get(cur, i) + sep +
list_get(cur, i + 1)) }
      else { bind nxt = list_push(nxt, list_get(cur, i)) }
      bind i = i + 2
    }
    bind cur = nxt
  }
  return list_get(cur, 0)
}
"
bind MAIN = "use \"plat.ling\"

bind start = do {
  println(p_name(), \": \", p_join([\"a\", \"b\", \"c\", \"d\", \"e\"], \"-\"))
}
"

fn run_variant(label, plat_text, main_text) {
  write_file("out/" + label + "/plat.ling", plat_text)
  write_file("out/" + label + "/main.ling", main_text)
  run_process("ling", ["run", "out/" + label + "/main.ling"], 30, "out/" + label +
".log")
  print(str_trim(read_file("out/" + label + ".log")))
}
```

```
bind start = do {
  run_variant("desktop", PLAT, MAIN)
  # the generated OS entry: the same text with exactly one use line swapped
  bind main_os = str_replace(MAIN, "plat.ling", "plat_os.ling")
  write_file("out/os/plat_os.ling", PLAT_OS)
  write_file("out/os/main_os.ling", main_os)
  run_process("ling", ["run", "out/os/main_os.ling"], 30, "out/os.log")
  print(str_trim(read_file("out/os.log")))
}
```

```
OUTPUT
desktop: a-b-c-d-e
kernel-style: a-b-c-d-e
```

Both backends print the same two lines. The point is what did not change: `main` is byte-for-byte identical apart from the one `use` line, yet the two runs went through different implementations of `p_join`. Notice also why the kernel version joins pairwise instead of appending one piece at a time. Repeated `bind out = out + piece` copies the growing string every time, and on a runtime that never frees, that is both slow and a permanent memory leak. Halving the list each pass produces `n - 1` strings in total but only `log n` of them are ever large. *Architecting Large Apps* measures the same effect on the desktop JIT.

> TIP
>
> Put the platform split at the lowest level where behaviour really differs (polygon fill, images, text input, file dialogs) and give every function there the same prefix (`p_`). The upper layers then stay ignorant of the target, and the generator script stays a one-line substitution.

Living with a runtime that frees nothing

LingModeler's `plat_os.ling` contains a function worth quoting, `p_idle_frame`, because it is the canonical answer to "nothing is ever freed". Its comment: "The kernel runtime frees nothing yet, so an unchanged screen is not rebuilt. Returns true when nothing happened since the last drawn frame (no pointer move, button, wheel, key or animation); the caller then only presents." The function compares the mouse position and buttons with the values saved in the `st_*` store, checks keys and playing animations, and keeps drawing "two more frames so release edges and hover states settle" after any change. The desktop version is `fn p_idle_frame() { return false }`, so the desktop always redraws.

Three habits follow for kernel-bound code:

1. **Do not allocate on idle frames.** Detect "nothing changed" cheaply and skip the whole draw.
2. **Keep state in the store.** `st_set` / `st_get` and the list helpers (`st_lpush`, `st_lget`, `st_llen`) overwrite in place; fresh lists each frame do not.
3. **Watch the number.** LingModeler prints `os_info("memory")` every 50 frames from `p_frame_hook`, and the desktop hook is a no-op. Build the same hook from day one.

`os_info` is a kernel-runtime builtin; the desktop runtime does not have it, which is exactly why it is called only from `plat_os.ling`.

The shipped apps and their targets

LingOS builds one image per kernel target. Each is a directory under `LingOS/kernel/` containing a `Ling.toml` and a `main.ling`:

```
OUTPUT
kernel/x86_64              live kernel
kernel/x86_64-installer    installer
kernel/x86_64-wm           graphics desktop (window manager, dock, built-in apps)
kernel/x86_64-lingmodeler  desktop with LingModeler built in
kernel/x86_64-lingdev      desktop with LingDev built in
kernel/rpi                 Raspberry Pi kernel
kernel/rpi-lingstation, rpi-lingdev, rpi-lingmodeler, rpi-studio ...
```

The desktop kernel's manifest is the best documentation of how an app is bundled. This excerpt is from `kernel/x86_64-wm/Ling.toml`, shortened:

```
[project]
name = "lingos-wm-x86_64"
entry = "main.ling"
kind = "kernel"
graphics = true

[app.lingstation]
entry = "../../../LingStationX/ling/main.ling"
root = "../../../LingStationX"
includes = ["assets/font.ttf", "assets/fonts/DaemonSans.ttf", "!assets/templates",
"projects/FunKey/project.json"]

[app.lingmodeler]
entry = "../../../LingModeler/ling/main_os.ling"
root = "../../../LingModeler"
includes = ["assets/font.ttf", "examples/*.lmesh", "examples/*.ling"]
```

`kind = "kernel"` makes it a bare-metal build and `graphics = true` asks the bootloader for a linear framebuffer. Each `[app.*]` section names an entry (the app's own `main.ling`, or the generated `main_os.ling` where a platform layer exists), a `root` that relative paths resolve against, and `includes` globs of files embedded into the image under `apps/<name>/`. LingDev's entry is its `main_os.ling` and LingModeler's is its generated one, matching what the previous section described.

The manifest also shows how a multi-process program is faked inside a single address space: the wallet's node, wallet daemon and miner are separate `iso_*` programs listed as extra `[app.iso_node]` entries and started with `thread_spawn`, which the comments say "runs them as green threads". If you have read *Concurrency and Isolates*, that is the same API with a different scheduler under it.

For a single app you do not need the whole desktop: LingModeler's README shows `ling build kernel/x86_64-lingmodeler --platform kernel --out dist` for an image that includes only it, and a `bundle_app = "lingmodeler:../../../LingModeler/ling/main_os.ling"` key in that manifest. On the host side, `ling build` also accepts `--platform lingos` (a ring-3 ELF app, `dist/apps/<name>.elf`) and `--platform rpi` or `lingos-rpi`. We did not run any of these builds: an end-to-end native build was blocked by a linker failure in this environment even on the desktop.

What each app targets, and what is missing

From the READMEs and the kernel sources, in the three cases the book's other chapters cover as case studies:

- **LingStationX** is built into the x86_64 desktop (`kernel/x86_64-wm`) and the Raspberry Pi studio images. In the kernel the `daw_*` builtins are served by `lingdaw.rs`, which holds the same `project.json` model as the desktop engine and answers reads and simple edits so the Arranger and track list show a real project. Its own header comment lists what was not there when written: instrument and effect DSP, audio output ("the transport clock runs, so the playhead moves, but nothing sounds"), audio samples, note editing and undo. The file also contains a section headed "step 2: sound", so the situation may have moved on; we did not check. Separately, LingOS has its own built-in LingStation tracker window driven by `ling-osdaw`, which does make sound through an AC'97 driver.
- **LingDev** runs as `main_os.ling` with `plat_os.ling`: software renderer, no audio, read-only files. Its project folders (scenes, scripts, levels) are embedded selectively through `includes`, with the complete projects expected on a data partition.
- **LingModeler** runs as the generated `main_os.ling`, with scanline polygons and image features reported as unavailable. Its README shows screenshots from QEMU at 1920 by 1080 (x86-64) and from QEMU `raspi3b` at 1280 by 720 (Pi), regenerated by `tools/lingos_qemu_test*.sh`.

For the stories of how these applications are structured, see *Case Studies: Applications Built in Ling*. For the architecture patterns that make the platform split cheap, see *Architecting Large Apps*.

Testing without booting

You do not need to boot a kernel to gain confidence in kernel-bound work, and the repository's own practice shows how:

1. **Host tests for freestanding crates.** `ling-osdaw` and `ling-llm` are `no_std` libraries with ordinary `cargo test` suites; their authors test on the host and let the kernel call the same functions.
2. **Cross-compile checks.** `cargo check --target x86_64-unknown-none` (and `aarch64-unknown-none`) is a fast, dependency-free proof that nothing in your crate reaches for `std`.
3. **Run the app on the desktop interpreter.** The shipped apps run with `ling run --interp ling/main.ling`, because that is the backend whose semantics the kernel copies. Their test harnesses drive the UI with scripted input arguments (`--drive`, `--do`, `--shot`) so screenshots can be compared headlessly.

4. **Boot in an emulator, headless.** `LingOS/live/boot-test.sh` boots the x86_64 ISO and the aarch64 `kernel8.img` in QEMU, captures their serial logs, and asserts on markers a healthy boot must print in order. This is the repository's CI check; we did not run it. If you build your own kernel target, add your app's start-up banner to the marker list.

Where this leaves you

LingOS is best treated as what it says it is: a serious, fast-moving research operating system whose strongest parts are the freestanding Rust engines and the platform-layer discipline of the apps, and whose gaps (processes, audio in some apps, newer Pi hardware, signed packages) are documented by its own authors. For a professional project, the practical advice is to adopt the *technique* first: keep your Ling app portable behind a platform layer, run it on `--interp`, budget for a runtime that never frees, and treat the kernel as an additional target rather than your only one.

Chapter summary

- LingOS is a Rust kernel plus Ling desktop and apps; apps are compiled into the kernel image and are not yet separate processes. Its README lists limits, and they are real.
- `ling-osdaw`, `ling-llm` and `ling-user` are `no_std`: all three type-check for `x86_64-unknown-none`, and `ling-osdaw` passed its 14 host tests and rendered a song to 1.8 MB of WAV in a plain Rust program. Model inference in `ling-llm` was not run here.
- The kernel runs `.ling` two ways: ahead-of-time compiled apps through `lingrt`'s `ling_builtin`, and a built-in compiler plus MIR interpreter for the Terminal.
- Kernel semantics follow the host interpreter, not the JIT (notably truthiness of empty text and lists), and the kernel runtime frees nothing.
- Build one app for two platforms with a platform layer (`plat.ling` and `plat_os.ling`) and a generator that swaps a single `use` line; keep the split at the lowest level where behaviour differs.
- Skip redraws on idle frames, keep state in `st_*`, and print the memory figure periodically.
- Kernel manifests use `kind = "kernel"`, `graphics`, and `[app.name]` sections with `entry`, `root` and `includes`.
- Test on the host, cross-check with `cargo check --target`, and use the repository's headless QEMU markers for boots.

Exercises

1. Write a platform-layer pair for a tiny app that needs a function `p_clock()`: on the desktop it returns `time_now()`, on a "kernel" variant it returns a counter held in the `st_*` store that the main loop advances by one each frame. Generate the OS entry the same way LingModeler does and print the first three values from each variant.
2. A freestanding Rust crate of yours needs `Vec` and `String` but must also run on the host. Describe the three changes that make it `no_std`-compatible and the one command that proves it.
3. Your kernel-bound Ling app builds a status line every frame with `bind line = line + part` over 40 pieces and draws it. Name two things that make this worse on LingOS than on the desktop, and two fixes.

Answers

1. `plat.ling`: `fn p_clock() { time_now() }`. `plat_os.ling`: `fn p_clock() { bind t = st_get("tick", 0) st_set("tick", t + 1) return t }` (the host-run text of both can be generated from string constants exactly as in the chapter's program). `main.ling` begins with `use "plat.ling"` and calls `p_clock()` three times in `start`; `main_os.ling` is the same text with `plat.ling` replaced by `plat_os.ling`. The kernel-style variant prints `0`, `1`, `2`; the desktop variant prints three increasing times. `st_get`/`st_set` exist on the stock desktop runtime too, so both variants run on the host.
2. Add `#![no_std]` and `extern crate alloc;`; import `alloc::vec::Vec` and `alloc::string::String` instead of the `std` prelude versions; replace `f32::sin`-style calls with the `libm` crate (`libm::sinf`) because `core` has no floating-point maths. To prove it, `rustup target add x86_64-unknown-none` and run `cargo check --target x86_64-unknown-none`. Keep `#[cfg(test)]` code behind `std` or put tests in `tests/`, as `ling-osdaw` does.
3. Worse because the kernel runtime frees nothing, so each frame's 40 intermediate strings stay allocated forever, and because repeated appending copies a growing string (quadratic work). Fixes: skip the redraw when nothing changed (the `p_idle_frame` pattern), and build the line once per change rather than per frame, joining pieces pairwise (or with `list_join` where it exists) instead of appending one by one.

CHAPTER 11

Architecting Large Apps

What you'll learn

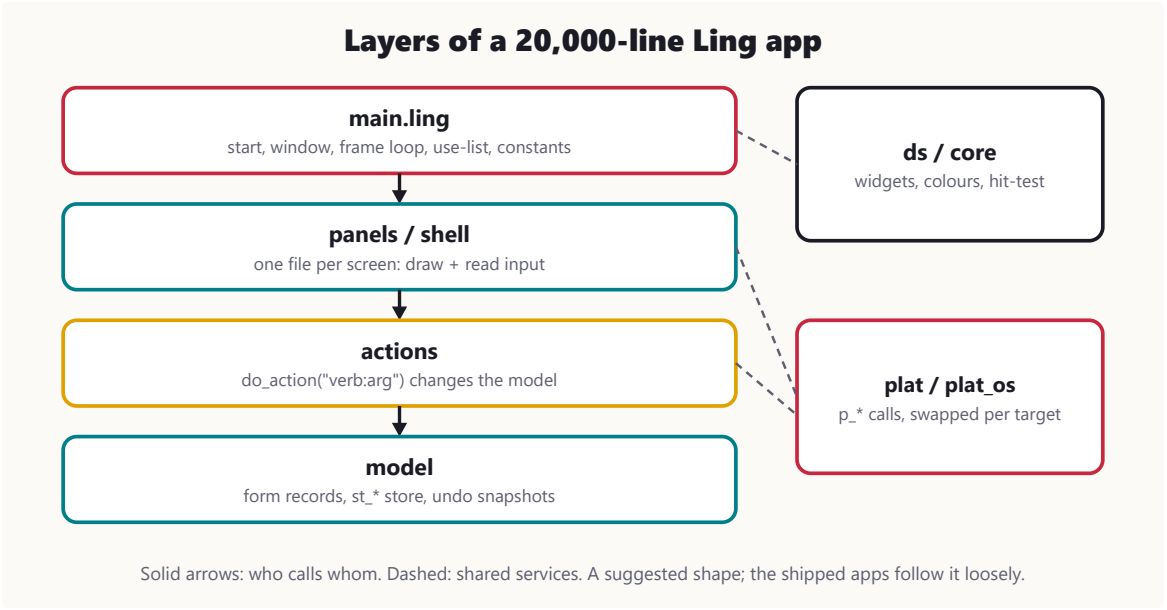
A 300-line Ling program can be written any way you like. A 20,000-line one cannot. This chapter collects the patterns that keep a large Ling application understandable, and the traps that appear only at that scale: a global namespace shared by dozens of files, no assignment, no mutable records, a JIT that hides errors, and a runtime where one innocent line can be a thousand times slower than it looks.

The raw material is real. Three desktop applications written entirely in Ling were read for this chapter: LingStationX, a digital audio workstation (25,894 lines in 60 files), LingDev, a game editor (28,358 lines in 33 files plus a `rt/` runtime folder), and LingModeler, a 3D and vector modelling tool (16,015 lines in 32 files). Line counts are from `wc -l` over each `ling/` folder. Their structure is described from their source and developer notes; every pattern we recommend beyond that was written as a small program and run on both backends, and the output shown is real. *Case Studies: Applications Built in Ling* tells the stories of these programs. Here we extract the engineering.

You will learn how to lay out files, where to keep state, how to route events and implement undo, how to persist settings and log, how to find and fix performance cliffs, how to test a program that has no assertion builtin, and how to ship it.

The shape of a large Ling app

All three applications converge on the same layering, which was never imposed by the language, only discovered the hard way.



A suggested shape, observed in the shipped apps. Calls go down; widgets and the platform layer are shared services.

- `main.ling` is short (284 lines in LingModeler, 614 in LingStationX). It holds the colour and layout constants, a long list of `use` lines, `start`, the window setup and the frame loop. In LingModeler the list runs from `plat.ling` through core, design system, icons, math, JSON, mesh, primitives and operations, views, formats, rig and pose, to `shell.ling`, `modals.ling`, `panels.ling` and `actions.ling` last.
- **A design-system layer** (`ds.ling`, `core.ling`, `icons.ling`): widgets (`u_button`, `u_hslider`), drawing helpers (`g_rect`, `g_text`), colours, hit testing. It knows nothing about the application's data.
- **Panels** (`shell.ling`, `mixer.ling`, `timeline.ling`, `view3d.ling`): each screen is a file that draws and reads input.
- **Actions** (`actions.ling`): the one place where user intent becomes a change to the model.
- **The model** (`mesh.ling`, `project.ling`): the data, and the functions that change it.
- **A platform layer** (`plat.ling` and `plat_os.ling`): everything that differs by target, covered in *Ling OS and Embedded*.

The order of `use` lines is a dependency order, and it is yours to keep. LingDev goes one step further: its `rt/` folder (18,502 lines, files named `rt_data`, `rt_cam`, `rt_cut`, `rt_game` …) is the game *runtime*, used both by the editor's preview and by the games it exports, so the editor files layer on top of a runtime that stands alone.

What `use` really gives you

The module system is deliberately thin, and *Modules and Project Structure* covers its mechanics. What matters for architecture is what we confirmed again on the installed `ling` 2030.1.53, on both backends:

Behaviour	Result
`use "util.ling"`	everything in that file becomes global: functions, forms, and top-level `bind` constants
paths	relative to the file that contains the `use`; `"../util.ling"` and `"sub/geom.ling"` work; running from another working directory works
a file used twice, or two files that `use` each other	tolerated, no error
a function in a used file reading a constant defined in the main file	works
two files defining the same function name	no error; the **last loaded wins**, and even functions inside the earlier file call the later one
`mod name { ... }`	`name::f(x)` works for self-contained functions, but a function inside the block that calls a sibling fails (`unknown function` on the interpreter, `()` on the JIT)

The last two rows decide the architecture. There are no namespaces you can rely on, so **prefix everything**. The apps use short file prefixes (`me_` for mesh editing, `fi_` for file I/O, `tl_` for the timeline, `v3_` for the 3D view, `ra_` for raster, `tx_` for textures, `u_` and `g_` for widgets and graphics, `p_` for the platform layer) and treat a collision as a bug that the language will never report. Avoid `mod`; use files. Keep generic names like `name` or `update` out of your global space.

> WARNING
>
> LingStationX's developer notes list reserved words as the first trap: every spelling of every keyword in every supported language is reserved everywhere, so a variable called `si` (Spanish `if`) fails with `expected LBrace, got Ident`. When a baffling parse error appears in an innocent line, suspect a reserved word in another language.

State: records, the store, and where each belongs

A Ling function sees top-level constants but not its caller's locals, cannot mutate anything it is handed, and records are immutable. State therefore has to live somewhere on purpose. There are three homes.

1. A record threaded through the program is the right tool for models that fit in your head: you write pure functions that return a new state, and history comes almost for free. This program is a complete small editor model with undo and redo. The commands are a `choose`, the state is two `form`s, and the whole reducer is a single `match`.

```
form Doc { title, items, }
form App { doc, past, future, }

choose Cmd {
  Add(text),
  Remove(index),
  Rename(title),
  Undo,
  Redo,
}

fn without(xs, skip) {
  bind out = []
  bind i = 0
  while i < len(xs) {
    if i != skip { bind out = list_push(out, list_get(xs, i)) }
    bind i = i + 1
  }
  out
}

fn edit(app, doc) {
  App(doc, list_push(app.past, app.doc), [])
}

fn last_of(xs) { list_get(xs, len(xs) - 1) }
fn drop_last(xs) { without(xs, len(xs) - 1) }

fn update(app, cmd) {
  match cmd {
    Add(t) => edit(app, Doc(app.doc.title, list_push(app.doc.items, t)))
    Remove(i) => edit(app, Doc(app.doc.title, without(app.doc.items, i)))
    Rename(t) => edit(app, Doc(t, app.doc.items))
    Undo() => {
      if len(app.past) == 0 { app } else {
        App(last_of(app.past), drop_last(app.past), list_push(app.future, app.doc))
      }
    }
```

```
    Redo() => {
      if len(app.future) == 0 { app } else {
        App(last_of(app.future), list_push(app.past, app.doc), drop_last(app.future))
      }
    }
    _ => app
  }
}

fn show(app) {
  println(app.doc.title, " ", app.doc.items, "  undo:", len(app.past), " redo:",
len(app.future))
}

bind start = do {
  bind app = App(Doc("Untitled", []), [], [])
  bind cmds = [Add("intro"), Add("body"), Rename("Report"), Remove(0), Undo(), Undo(),
Redo(), Add("end")]
  bind i = 0
  while i < len(cmds) {
    bind app = update(app, list_get(cmds, i))
    show(app)
    bind i = i + 1
  }
}
```

```
OUTPUT
Untitled [intro]  undo:1 redo:0
Untitled [intro, body]  undo:2 redo:0
Report [intro, body]  undo:3 redo:0
Report [body]  undo:4 redo:0
Report [intro, body]  undo:3 redo:1
Untitled [intro, body]  undo:2 redo:2
Report [intro, body]  undo:3 redo:1
Report [intro, body, end]  undo:4 redo:0
```

Both backends agree. Undo here is nothing but keeping the previous `Doc` in a list, since records are values; any new edit empties `future`, which is the standard redo rule. Three details are load-bearing: the two `form` declarations end with a trailing comma (a typed last field would swallow the brace), nullary variants are matched as `Undo()`, and the constructor of the `choose` is `Undo()` too. The cost is that every layer must pass the state down and return it up, which is pleasant at 200 lines and heavy at 20,000.

2. The `st_*` store is what the large apps chose. It is a global key-value store with typed helpers: `st_set(key, value)`, `st_get(key, default)`, `st_has(key)`, `st_add`, `st_del`, and list operations `st_lpush`, `st_lget`, `st_lset`, `st_llen`, `st_ldel`, `st_lclear`. Any function can read or write it, which removes the threading problem. LingStationX's developer notes state the rule that produced this: functions cannot mutate caller variables and lists are values, so "all shared state lives in the native store". The store works on the stock `ling` too; we checked `st_set`, `st_get` with a default, `st_has` and the list helpers on both backends.

The price is that you now have a global, untyped, string-keyed namespace. Treat it like one: prefix keys by module (`nd_`, `pf_`, `ix_`), keep the list of keys for each module in the module's header comment, and write small accessor functions rather than scattering string literals. A typo in a key does not fail; `st_get("itmes", [])` quietly returns the default.

3. Files are for what must survive: projects, settings, caches. Keep serialization next to the model it describes and make loaders tolerant, as the settings section below shows.

> TIP
>
> Use a record for state that is local to one subsystem and benefits from history (an editor buffer, a game board). Use the store for state that many files must see (the selected track, the current tab, preferences). Never use the store as a hidden argument list between two functions that could take parameters.

Event routing: one function with string actions

The three apps route every user intent through a single function, `do_action(a)`, where `a` is a short string in `verb:argument` form. Buttons, menus, keyboard shortcuts and tests all call it. LingModeler's version checks `str_starts_with(a, "mode:")`, `"add:"`, `"open:"`, `"saveas:"` and so on, then does the work and returns. This has four advantages that suit Ling's limits: no closures are needed (closures in variables are unreliable on the JIT), the set of things the program can do is greppable, a menu item is a string, and a test can drive the app by name. Undo also becomes easy to enforce: every mutating action calls the undo function first, in one place.

The next program shows the pattern with store-backed undo. State lives in the store (`items`), snapshots are copies of the list under numbered keys, and `do_action` is the only door:

```
# Event routing with string actions and snapshot undo in the st_* store.

fn items() { st_get("items", []) }

fn undo_push() {
  bind n = st_get("undo_n", 0)
  st_set("u" + str(n), items())
  st_set("undo_n", n + 1)
  st_set("redo_n", 0)
}

fn undo() {
  bind n = st_get("undo_n", 0)
  if n < 1 { return "nothing to undo" }
  bind r = st_get("redo_n", 0)
  st_set("r" + str(r), items())
  st_set("redo_n", r + 1)
  st_set("items", st_get("u" + str(n - 1), []))
  st_set("undo_n", n - 1)
  return "undone"
}

fn redo() {
  bind r = st_get("redo_n", 0)
  if r < 1 { return "nothing to redo" }
  bind n = st_get("undo_n", 0)
  st_set("u" + str(n), items())
  st_set("undo_n", n + 1)
  st_set("items", st_get("r" + str(r - 1), []))
  st_set("redo_n", r - 1)
  return "redone"
}

fn do_action(a) {
  if str_starts_with(a, "add:") {
    undo_push()
    st_set("items", list_push(items(), str_slice(a, 4, len(a) - 4)))
    return "added"
```

```
  }
  if str_starts_with(a, "clear") {
    undo_push()
    st_set("items", [])
    return "cleared"
  }
  if a == "undo" { return undo() }
  if a == "redo" { return redo() }
  return "unknown action: " + a
}

bind start = do {
  bind script = ["add:kick", "add:snare", "clear", "undo", "undo", "redo", "bogus",
"undo", "undo", "undo"]
  bind i = 0
  while i < len(script) {
    bind msg = do_action(list_get(script, i))
    println(list_get(script, i), " -> ", msg, "   items=", items())
    bind i = i + 1
  }
}
```

```
OUTPUT
add:kick -> added   items=[kick]
add:snare -> added   items=[kick, snare]
clear -> cleared   items=[]
undo -> undone   items=[kick, snare]
undo -> undone   items=[kick]
redo -> redone   items=[kick, snare]
bogus -> unknown action: bogus   items=[kick, snare]
undo -> undone   items=[kick]
undo -> undone   items=[]
undo -> nothing to undo   items=[]
```

Output is identical on `--interp`. Note the snapshot cost: each undo step stores a full copy of what it protects. LingModeler does exactly this on a larger scale, snapshotting every scene key under `u0_`, `u1_`… and capping history at 40 steps, shifting the oldest out when the cap is reached. Cap yours; an unbounded history is a memory leak with a menu item.

The scripted driver is the other half of the pattern. The apps accept `--do "frame,action;frame,action"` on the command line (LingModeler's README also describes `--drive` for synthetic mouse input, `--shot N` to save a screenshot at frame N and exit, and `--mode` and `--size` to open at a known state; LingStationX adds `--tab`). Arguments arrive through `get_args()`, which returns the whole command line including the executable and `run` and the script path, so a helper scans for its flag and takes the next element. An automated test can then say "at frame 10 do `add:kick`, at frame 30 do `undo`, at frame 40 take a screenshot" and compare the pixels.

Settings that survive upgrades

The apps keep settings in a plain `key=value` text file next to the program (`settings.txt`), with repeated keys for lists (`recent=...`). The format is deliberately the same shape as the line-based reader `ling build` uses for `Ling.toml`: it is trivially diffable, trivially hand-editable, and trivially parseable with the text builtins. LingStationX's loader reads the file, splits lines, cuts each at the first `=`, and dispatches on the key. Three rules make such a loader safe, and this program demonstrates all of them, including recovering from a damaged file:

```
form Settings { theme, volume, recent, }

fn defaults() { Settings("dark", 0.8, []) }

fn serialize(s) {
  bind out = "theme=" + s.theme + "\nvolume=" + str(s.volume) + "\n"
  bind i = 0
  while i < len(s.recent) {
    bind out = out + "recent=" + list_get(s.recent, i) + "\n"
    bind i = i + 1
  }
  out
}

fn parse(text) {
  bind s = defaults()
  bind lines = str_split(text, "\n")
  bind i = 0
  while i < len(lines) {
    bind line = str_trim(list_get(lines, i))
    bind eq = str_find(line, "=")
    if eq > 0 {
      bind key = str_slice(line, 0, eq)
      bind val = str_slice(line, eq + 1, len(line) - eq - 1)
      if key == "theme" { bind s = Settings(val, s.volume, s.recent) }
      if key == "volume" { bind s = Settings(s.theme, to_number(val), s.recent) }
      if key == "recent" { bind s = Settings(s.theme, s.volume, list_push(s.recent,
val)) }
    }
    bind i = i + 1
  }
  s
}

fn load(path) {
  if file_exists(path) { parse(read_file(path)) } else { defaults() }
}
```

```
bind start = do {
  bind path = "out/settings.txt"
  bind s = load(path)
  println("first run : ", s)
  bind s = Settings("light", 0.55, ["a.proj", "b.proj"])
  write_file(path, serialize(s))
  println("saved     :", str_replace(read_file(path), "\n", " | "))
  println("reloaded  : ", load(path))
  write_file(path, "theme=solar\nbogus line\nvolume=oops\n")
  println("damaged   : ", load(path))
}
```

```
OUTPUT
first run : Settings { theme: dark, volume: 0.8, recent: [] }
saved     :theme=light | volume=0.55 | recent=a.proj | recent=b.proj |
reloaded  : Settings { theme: light, volume: 0.55, recent: [a.proj, b.proj] }
damaged   : Settings { theme: solar, volume: 0, recent: [] }
```

1. **Start from defaults**, never from "empty". A missing file, a missing key and an unknown key are all normal. The loader begins with `defaults()` and only overrides what it understands, so a settings file from a newer or older version cannot break start-up.
2. **Ignore what you cannot parse.** The `bogus line` was skipped. But look at the last field: `volume=oops` became `0`, because `to_number("x")` is `0` rather than an error. Ling's failure convention is soft, so *validate after parsing* (clamp volume to a legal range, reject zero where zero is nonsense) instead of trusting the conversion.
3. **Write whole files, and write at moments you control.** `write_file` creates parent directories and replaces the file. Save on change events, not only on exit, because a crashed program never reaches its exit code. For files that must not be half-written, write to a temporary name first and then `rename_file` it into place.

Logging when there is no stderr

Ling has no logging library and no builtin for printing to standard error, and `append_file` does not exist (it appears in some capability tables, but the runtime has no such builtin; the interpreter says `unknown function 'append_file'`). The shipped apps solve this at two levels.

First, the **launcher** captures everything. LingDev's `run.bat` runs `ling run --interp ling\main.ling %* > lingdev.log 2>&1` and, if the exit code is non-zero, shows the last 40 lines of the log. Everything the program prints, including the runtime's own error tracebacks, ends up in one file. Second, inside the program, a **leveled logger** prints a tagged line and also keeps a small ring buffer in the store, so a crash report or an in-app console can show recent history without reading a file:

```
bind LOG_PATH = "out/app.log"
bind LOG_KEEP = 4

fn log(level, msg) {
  if st_get("log_level", 1) <= level {
    bind names = ["DEBUG", "INFO", "WARN", "ERROR"]
    bind line = list_get(names, level) + " " + msg
    println(line)
    st_lpush("log_ring", line)
    while st_llen("log_ring") > LOG_KEEP { st_ldel("log_ring", 0) }
  }
}

fn log_flush() {
  bind lines = []
  bind i = 0
  while i < st_llen("log_ring") {
    bind lines = list_push(lines, st_lget("log_ring", i))
    bind i = i + 1
  }
  write_file(LOG_PATH, list_join(lines, "\n") + "\n")
}

bind start = do {
  st_set("log_level", 1)
  log(0, "not written: below threshold")
  log(1, "project opened")
  log(2, "autosave is late")
  log(3, "could not read settings.txt")
  log(1, "closing")
  log(1, "one more line")
  log_flush()
  println("--- file keeps the last " + str(LOG_KEEP))
  print(str_trim(read_file(LOG_PATH)))
}
```

```
OUTPUT
INFO project opened
WARN autosave is late
ERROR could not read settings.txt
INFO closing
INFO one more line
--- file keeps the last 4
WARN autosave is late
ERROR could not read settings.txt
INFO closing
INFO one more line
```

Identical on `--interp`. The level threshold lives in the store, so a `--debug` flag or a settings checkbox flips it without recompiling. Flush the ring from your crash path and from `p_frame_hook`-style periodic hooks; the LingStationX tree even carries a `lingstation_crash.log` next to `lingstation.log`, the result of exactly this habit.

> NOTE
>
> The default backend never reports a runtime error, so the first line of defence in a large app is to **run it on** `--interp`, which is what all three READMEs tell you to do. A traceback that says `0: start`, `1: expect` is worth more than any log line the JIT would never have written.

Performance: the cliffs

At this scale, performance is rarely about clever algorithms. It is about a handful of cliffs, and the cliffs differ by backend. This benchmark exercises four operations at sizes where the differences show. The `depth` line is a trick: the JIT optimizes aggressively and can fold a call on constant arguments, so the argument is made to depend on `time_now()`.

```ling
fn fib(n) {
  if n < 2 { return n }
  return fib(n - 1) + fib(n - 2)
}

fn build_list(n) {
  bind xs = []
  bind i = 0
  while i < n {
    bind xs = list_push(xs, i)
    bind i = i + 1
  }
  len(xs)
}

fn build_text(n) {
  bind s = ""
  bind i = 0
  while i < n {
    bind s = s + "x"
    bind i = i + 1
  }
  len(s)
}

fn build_parts(n) {
  bind parts = []
  bind i = 0
  while i < n {
    bind parts = list_push(parts, "x")
    bind i = i + 1
  }
  len(list_join(parts, ""))
}

fn lap(label, result, t0) {
  println(label, " = ", result, "  ", round((time_now() - t0) * 1000), " ms")
}
```

```
bind start = do {
  bind t = time_now()
  bind depth = 32 + floor(time_now() * 0)
  lap("fib(32)          ", fib(depth), t)
  bind t = time_now()
  lap("list_push x40000 ", build_list(40000), t)
  bind t = time_now()
  lap("text += x40000   ", build_text(40000), t)
  bind t = time_now()
  lap("parts + join x40000", build_parts(40000), t)
}
```

On the default JIT, then on `--interp`, on one machine (timings vary from run to run; the ratios are the lesson):

```
OUTPUT
JIT:
fib(32)           = 2178309  14 ms
list_push x40000  = 40000  4 ms
text += x40000    = 40000  1101 ms
parts + join x40000 = 40000  15 ms

INT:
fib(32)           = 2178309  1827 ms
list_push x40000  = 40000  8 ms
text += x40000    = 40000  39 ms
parts + join x40000 = 40000  11 ms
```

Read it carefully, because the two backends have opposite pathologies:

- **Compute.** The JIT ran `fib(32)` over 100 times faster than the interpreter. For numeric loops, the JIT is the reason to ship natively.
- **Repeated text append is quadratic on the JIT.** Building a 40,000-character string one character at a time took 1.1 seconds, against 39 milliseconds on the interpreter. At 50,000 characters we saw about three to four seconds. Each `s + "x"` copies the whole string. The fix is general and cheap: collect the pieces in a list and `list_join` once, which took 15 ms here. A separate run joining 20,000 short words (about ten characters each) took about 5

seconds when appended one by one on the JIT, 44 ms with LingOS-style pairwise halving, and 2 ms with `list_join`. Hot paths that build text (log lines, serializers, exporters, `.glb` or SVG writers) should never append in a loop.

- **List growth is fine in both**, so `list_push` in a `while` loop is not the problem; rebinding is.

Practical rules that follow from this and from the apps:

1. **Measure with `time_now()` around the suspect, on the backend you ship.** The apps ship on the interpreter (their desktop builds embed it, see below) and so should be profiled there. If you ship natively, profile the JIT: the cliff you hit may not exist on the other backend.
2. **Redraw only when something changed.** An immediate-mode UI that rebuilds every pixel every frame is the single largest cost in a Ling app. The idle-frame check from *Ling OS and Embedded* (compare mouse, keys and animation state against the store; skip the draw) transfers to the desktop.
3. **Cache derived data under a key.** If a layout or a mesh only changes on certain actions, compute it in the action and store the result under a key, not in the frame loop.
4. **Prefer `while` to `for` when you accumulate.** `for` gives each iteration a fresh copy of the scope, so a `bind` inside is lost.
5. **Compare fields, not records.** `==` on lists and records is always false, so equality tests silently degrade to "changed", which triggers redraws and writes.
6. **Watch recursion and giant functions.** The compiler has `--O0` to `--O3` flags for the JIT; if compile time blows up on a very large function, `--O0` is the escape hatch. And note what the LingModeler README reports for the *native* AOT build: the Cranelift backend panicked on its 750-function program ("a toolchain bug, not an app error; every module compiles on its own"), so the app ships as an interpreter-embedded launcher instead. LingStationX's notes likewise say the JIT currently fails on that whole program and to always use `--interp`. Knowing which backend your app can really run on is part of the architecture.

Testing a program with no assert

Ling has no test framework and no assertion builtin, and `lingfu test` has a blind spot you must know about. We built a project with `lingfu new`, put a deliberately failing test in `tests/`, and ran it:

```
OUTPUT
Testing Ling project...
  test: <project>\tests\t_bad.ling
FAIL total off by one: got 32 want 33
✓ All tests passed!
```

`lingfu test` ran the file, which printed `FAIL`, and then reported success. It runs each test with `ling run`, which uses the JIT, and the JIT never fails at run time: unknown functions, bad arithmetic, missing files and failed matches all produce `()` and exit status 0. (The same file run with `ling run --interp` exits with status 1.) Also, every `.ling` file under `tests/` is run as a test, so a shared helper placed there fails with "no entry point"; keep helpers in `src/`.

The workable strategy is a small kit plus your own runner:

1. **A failing assertion must be an error the interpreter reports.** Call a function that does not exist. On `--interp` that raises `unknown function`, prints a traceback with the failing test's name, and exits with status 1.
2. **Tests are ordinary programs** with a `start`, one file per concern, using `use` to load the code under test.
3. **A runner runs each file under** `--interp` with `run_process`, which returns the child's exit code, and sums failures. Wire the runner's own exit into CI as you can (for example, by finishing with a deliberate error when `failed > 0`, or by having CI check the printed count).

The program below is self-contained: it writes a mini project to disk (`src/rules.ling`, `src/testkit.ling`, and two tests, one good and one bad), runs both under `--interp`, and reports.

```
# Writes a tiny project (src/ and tests/), then runs every test file under --interp.
bind RULES = "fn clamp_qty(q) { max(0, q) }
fn total(prices, qtys) {
  bind i = 0
  bind sum = 0
  while i < len(prices) {
    bind sum = sum + list_get(prices, i) * list_get(qtys, i)
    bind i = i + 1
  }
  sum
}
"

bind TESTKIT = "fn expect(name, got, want) {
  if got == want {
    println(\"ok   \", name)
  } else {
    println(\"FAIL \", name, \": got \", got, \" want \", want)
    test_failed()
  }
}
"

bind T_GOOD = "use \"../src/rules.ling\"
use \"../src/testkit.ling\"

bind start = do {
  expect(\"clamp low\", clamp_qty(0 - 5), 0)
  expect(\"clamp keep\", clamp_qty(7), 7)
  expect(\"total\", total([2, 3], [10, 4]), 32)
}
"

bind T_BAD = "use \"../src/rules.ling\"
use \"../src/testkit.ling\"

bind start = do {
  expect(\"total off by one\", total([2, 3], [10, 4]), 33)
```

```
}
"

bind start = do {
  write_file("proj/src/rules.ling", RULES)
  write_file("proj/src/testkit.ling", TESTKIT)
  write_file("proj/tests/t_good.ling", T_GOOD)
  write_file("proj/tests/t_bad.ling", T_BAD)
  write_file("proj/test.log", "")
  bind files = ["proj/tests/t_good.ling", "proj/tests/t_bad.ling"]
  bind failed = 0
  bind i = 0
  while i < len(files) {
    bind code = run_process("ling", ["run", "--interp", list_get(files, i)], 30,
"proj/test.log")
    println(list_get(files, i), " -> exit ", code)
    if code != 0 { bind failed = failed + 1 }
    bind i = i + 1
  }
  println("failed files: ", failed)
  println("--- log")
  print(str_trim(read_file("proj/test.log")))
}
```

```
OUTPUT
proj/tests/t_good.ling -> exit 0
proj/tests/t_bad.ling -> exit 1
failed files: 1
--- log
ok   clamp low
ok   clamp keep
ok   total
FAIL total off by one: got 32 want 33
error[runtime]: unknown function 'test_failed'
 --> proj/tests/t_bad.ling
  traceback (deepest call last):
    0: start
    1: expect
  hint: check the spelling, or `use` the module that defines it
```

The same output appears on both backends of the runner itself, because each child is started with `--interp` regardless. A few more layers belong in a serious strategy:

- **Pure logic first.** Mesh maths, serializers, parsers and rules are plain functions; test them in thousands of cases with the kit above. LingModeler's `json.ling`, `mathx.ling`, `objfmt.ling` and `gltf.ling` are all of this kind.
- **Scripted UI runs.** Use the `--do`, `--drive` and `--shot` arguments from the event-routing section to replay a session and capture a screenshot; LingStationX's notes describe `tools/shot.py` (screenshot with optional crop) and `tools/smoke.py` (opens every dialog, tab and menu) built on exactly that. Compare screenshots to golden files, and check for changed pixels rather than the whole image if layouts shift.
- **Smoke test every module compiles.** A parse error in a file nobody loads from the main path will otherwise surprise you at release.
- **Run tests on `--interp` and also once on the JIT** with a quick sanity pass, because the two backends genuinely differ (see *How Ling Runs*).

Release engineering

A release of a large Ling app has four moving parts: a manifest, a build matrix, a backend choice and a checklist.

The manifest. `Ling.toml` drives `ling build` and `lingfu`. LingStationX's manifest, trimmed, shows the keys that matter at this scale:

```
[spirit]
name = "ling-station"
kind = "ui"
version = "2030.1.4"
entry = "ling/main.ling"
portable = "ling"
targets = ["win", "mac", "lin", "kernel", "rpi"]
entry_os = "ling/main.ling"
includes = ["ling/*.ling", "scripts/*.ling", "assets/font.ttf", "assets/fonts/*",
"instruments/*.ling"]
```

`entry` is relative to the manifest (`lingfu new` writes a project whose entry is `src/start.ling`, so you must set it). `portable = "ling"` makes desktop builds run `ling/main.ling` on the interpreter from the folder beside the executable, which keeps a multi-folder `use` layout intact: that is how all three apps ship on Windows, Linux and macOS. `includes` lists resources copied next to the executable, or embedded with `--pack`. `targets` is the default platform list, and `entry_os` and `includes_os` define the lean variant for the kernel targets. The author's own comment in that manifest explains the sharpest edge: "kept on ONE line, which is what `ling build` reads". `ling build` reads the manifest with a line-based parser, not a TOML parser: keys must be `key = value` with a space before `=`, and arrays must be written on one line.

The build matrix. LingStationX drives its builds from one script (`tools/build.py win|lin|mac|web|lingos-x86|lingos-pi5|all`), run on each operating system's own CI runner, with an Inno Setup installer for Windows, a `.dmg` for macOS and a `.deb` plus portable archive for Linux, and a weekly workflow that publishes pre-releases. LingModeler and LingDev have their own `build_desktop` and `build_dist` scripts. They embed large optional assets by reference rather than bundling: LingStationX's manifest says its language models "are far too large to bundle" and users point the app at a `.gguf` file. Build one artifact per platform and never assume cross-compilation: a native `ling build` generates a Rust project and compiles it, so it needs a Rust toolchain, and even that could not be run end to end in this book's environment because of a local linker failure. Treat those commands as documented, not verified.

The backend. Decide deliberately between the three modes. The interpreter-embedded launcher (`lingfu build --no-aot`, or `portable`) is what the large apps ship, because it is the semantic reference and the only backend on which their 20,000-plus lines run. Plain `lingfu`

`build` defaults to native AOT through Cranelift, which is faster but, per the LingModeler README, panics on a 750-function program. Choose a mode by running your real app in it, not by reading about it.

A checklist, in the order we would run it:

1. Run the test runner on `--interp`; fix every failure.
2. Run the app once on the JIT; compare behaviour, since a divergence is a latent bug.
3. Smoke-test every module and every dialog with scripted input.
4. Confirm `Ling.toml`: version bumped, `entry` correct, `includes` complete, arrays on one line.
5. Build every target on its own OS; launch each artifact from a clean folder with no repository beside it, because a missing `includes` entry only shows there.
6. Check the log file path and that crash output is captured.
7. Record the `ling` version the build used. Behaviour has changed between releases (for example `str_to_hex`, `rand_seed` and `net_local_ip` exist only in newer builds and are silent `()` in older ones), and a user's installed `ling` can be older than yours.

> **TRY IT**
>
> Take any program of yours with more than three files. Give each file a prefix, move every platform-specific call behind a `p_` function, add the `do_action` router, run it under `--interp`, and add a runner that executes one test file per module. You will find at least one name collision and one silent `()` in the first hour.

Chapter summary

- Large Ling apps converge on layers: `main` (constants, `use` list, loop), design system, panels, actions, model, platform layer. The `use` order is the dependency order.
- `use` flattens everything into one namespace, including top-level binds; the last definition of a name silently wins, and `mod` blocks are unreliable. Prefix every function and every store key.
- Keep state in a record when it is local and benefits from history, and in the `st_*` store when many files share it. Never rely on the default for a mistyped key.
- Route every intent through `do_action("verb:arg")`; put undo, logging and tests at that single door; cap undo history.
- Settings are `key=value` text with defaults, tolerant parsing and validation after conversion. Remember that `to_number("oops")` is `0`.

- Logging relies on the launcher capturing stdout and stderr to a file, plus an in-store ring buffer; `append_file` does not exist.
- The JIT is over 100 times faster on compute and orders of magnitude slower on repeated text appends; collect pieces and `list_join`. Redraw only when something changed.
- `lingfu test` runs the JIT and reports success for failing tests; write failures as unknown-function errors and run each test file under `--interp` from your own runner.
- A release is a one-line-array `Ling.toml`, a per-platform build, a deliberate backend choice (interpreter-embedded launcher or native AOT), and a clean-machine launch.

Exercises

1. In the reducer program, add a `Move(from, to)` command that reorders two items and is undoable. Which function must change, and why does undo need no extra code?
2. Extend the string-action router with `rename:<title>` stored under the key `title`, undoable like `add:`. What does `undo_push` need to snapshot now, and what would break if you forgot?
3. A teammate's exporter builds a 200 KB SVG with `bind svg = svg + piece` in a loop and "gets slower every release". Diagnose it, fix it, and describe how you would prove the fix on the backend you ship.

Answers

1. Only `update` changes: add a `Move(a, b) => edit(app, Doc(app.doc.title, moved(app.doc.items, a, b)))` arm, with a helper `moved` that builds the new list. Undo already works because `edit` pushes the previous `Doc` onto `past`, and records are values, so the snapshot cannot be changed by later edits. No other code needs to know about `Move`.
2. `undo_push` must also snapshot the title: `st_set("ut" + str(n), st_get("title", ""))`, and `undo` and `redo` must restore it the same way (for symmetry, also save it in the redo slots). If you forgot, undo would restore the items but leave the new title, an inconsistent state that no test of `add:` alone would reveal. A good rule: snapshot every key an action can touch, and list those keys in one function that both push and restore use.
3. Repeated `svg + piece` copies the whole growing string on every iteration (quadratic), and it is dramatically worse on the JIT (about 1.1 s for 40,000 single-character appends against 39 ms on the interpreter in this chapter's benchmark). It "gets slower each release" because each release adds output. Fix: `bind parts = list_push(parts, piece)` in the loop and one

`list_join(parts, "")` at the end. Prove it by timing the exporter with `time_now()` before and after, on the backend you ship, with the biggest real document, and keep the timing as a regression check in the test runner.

PART

Recipes

Patterns for professional applications.

CHAPTER 12

Professional App Recipes

Thirty-two self-contained patterns for the plumbing every real application needs: configuration, logging, retries, caches, queues, validation, migrations, tokens, templating, internationalisation and more. Each recipe is a complete program that was run with `ling run` and, to prove portability, with `ling run --interp`; where the two backends gave different output the recipe was rewritten until they agreed, and the one exception is called out in its text. The printed output is exactly what the program produced.

How to read these recipes

The recipes share a style that comes straight from how Ling works. Skim this once and the code will look familiar.

- **State lives in records and is passed in and out.** There is no assignment, so each function takes the old value and returns the new one. `bind x = f(x)` rebinds.
- **Time is a parameter.** Recipes that need a clock take `now` as an argument so you can test them without sleeping. In production pass `time_now()` or `epoch_now()`.
- **Loops are `while`.** A `for` body gets its own scope each iteration, so it cannot accumulate into an outer variable.
- **Collections are lists.** There are no dictionaries, sets or sort builtin; the recipes build maps from `[key, value]` pairs and use explicit loops.
- **Failure is a value.** Functions return `ok(...)` or `bad(...)` and callers unwrap with `match`. A helper such as `fn payload(r) { match r { ok(v) => v, bad(e) => e } }` appears throughout.
- **Functions are called by name.** Function values passed to builtins or stored in lists work only on the interpreter, so recipes dispatch on names instead.

Three parse traps bit while these recipes were written; they are worth knowing before you write your own:

> **WARNING**
>
> A line that begins with `[` or `-` continues the previous expression. A list literal on its own line after a `}` or a call is read as an index (`error[parse]: expected RBracket`), and a bare `-1` after a loop is read as subtraction. Bind the value to a name first, or use `return`.

> **WARNING**
>
> `wait`, `spawn`, `stop`, `again`, `change` and several other words are reserved by the lexer even though they do nothing. `bind wait = 5` is a parse error; name it `delay`.

> **NOTE**
>
> `&&` and `||` do not short-circuit on the interpreter. Never write `i < len(xs) && xs[i] == x` when the second operand can fail; nest the `if`s.

Each recipe is tagged for its edition: `ed: I` marks an intermediate technique, `ed: A` an advanced one.

Config files with typed getters

Read an application's settings from a plain `key = value` file, with comments, blank lines and defaults.

```
// Config file: key = value lines, hash comments, typed getters with defaults.
fn parse_config(text) {
  bind lines = split(text, "\n")
  bind cfg = []
  bind i = 0
  while i < len(lines) {
    bind line = trim(lines[i])
    bind at = str_find(line, "=")
    if len(line) > 0 && !starts_with(line, "#") && at > 0 {
      bind key = trim(substr(line, 0, at))
      bind val = trim(substr(line, at + 1, len(line) - at - 1))
      bind cfg = list_push(cfg, [key, val])
    }
    bind i = i + 1
  }
  cfg
}

fn cfg_get(cfg, key, fallback) {
  bind i = 0
  bind found = fallback
  while i < len(cfg) {
    if cfg[i][0] == key { bind found = cfg[i][1] }
    bind i = i + 1
  }
  found
}

fn cfg_number(cfg, key, fallback) { to_number(cfg_get(cfg, key, to_str(fallback))) }
fn cfg_bool(cfg, key, fallback) { cfg_get(cfg, key, to_str(fallback)) == "true" }

bind start = do {
  write_file("app.conf", "# server settings\nhost = example.local\nport = 8080\ndebug =
true\n\nname = My App = v2\n")
  bind cfg = parse_config(read_file("app.conf"))
  println("host:    ", cfg_get(cfg, "host", "localhost"))
  println("port+1:  ", cfg_number(cfg, "port", 80) + 1)
  println("debug:   ", cfg_bool(cfg, "debug", false))
```

```
  println("name:    ", cfg_get(cfg, "name", "?"))
  println("timeout: ", cfg_number(cfg, "timeout", 30))
  remove_file("app.conf")
}
```

```
OUTPUT
host:    example.local
port+1:  8081
debug:   true
name:    My App = v2
timeout: 30
```

`parse_config` splits the file into lines and cuts each at the **first** `=` (`str_find`), so values may themselves contain `=` (see `name`). Settings are stored as a list of `[key, value]` pairs; `cfg_get` scans for a key and returns a fallback if it is absent. Typed getters convert on the way out. Variation: write the same file back with `list_join` to persist changes (see *Settings persistence*).

Logging with levels

A logger that drops messages below a threshold and keeps the rest to flush to a file in one write.

```
// Leveled logging: a Logger record, a threshold, and lines buffered for one write.
form Logger { level, lines, tick }

fn level_name(n) {
  if n == 0 { return "DEBUG" }
  if n == 1 { return "INFO" }
  if n == 2 { return "WARN" }
  "ERROR"
}

fn make_logger(level) { Logger(level, [], 0) }

fn log_at(lg, level, msg) {
  if level < lg.level { return lg }
  bind line = format("{} [{}] {}", lg.tick, level_name(level), msg)
  println(line)
  Logger(lg.level, list_push(lg.lines, line), lg.tick + 1)
}

fn log_flush(lg, path) {
  write_file(path, list_join(lg.lines, "\n") + "\n")
  Logger(lg.level, [], lg.tick)
}

bind start = do {
  bind lg = make_logger(1)
  bind lg = log_at(lg, 0, "cache warmed")
  bind lg = log_at(lg, 1, "server started")
  bind lg = log_at(lg, 2, "disk at 91 percent")
  bind lg = log_at(lg, 3, "request failed")
  bind lg = log_flush(lg, "app.log")
  println("file has ", len(split(trim(read_file("app.log")), "\n")), " lines")
  remove_file("app.log")
}
```

```
OUTPUT
0 [INFO] server started
1 [WARN] disk at 91 percent
2 [ERROR] request failed
file has 3 lines
```

The logger is a record, so every call returns a new `Logger`; nothing is global. Levels are numbers so comparison is a plain `<`. `format` fills each `{}` in turn. Buffering lines and writing once keeps the file consistent if the program dies between log calls. Pass a counter or `epoch_now()` as the first field to add timestamps; the counter here keeps the output reproducible.

Command-line options

Read `--name value` options with defaults, and print usage when a required one is missing.

```
// CLI options: cli_arg(name, default) reads "--name value" pairs after the script
path.
fn usage() { "usage: ling run tool.ling --name NAME [--count N] [--shout true]" }

bind start = do {
  bind name = cli_arg("name", "")
  bind count = to_number(cli_arg("count", "1"))
  bind shout = cli_arg("shout", "false") == "true"
  if name == "" {
    println(usage())
  } else {
    bind i = 0
    while i < count {
      bind msg = "hello, " + name
      if shout { bind msg = str_upper(msg) }
      println(msg)
      bind i = i + 1
    }
  }
}
```

```
OUTPUT
usage: ling run tool.ling --name NAME [--count N] [--shout true]
```

`cli_arg(name, default)` returns the value following `--name` on the command line and the default otherwise. With no arguments the program prints its usage line (above). Run it with options:

```
ling run r03.ling --name Ada --count 2 --shout true
```

```
OUTPUT
HELLO, ADA
HELLO, ADA
```

Options are forwarded on both backends. Every option is a name/value pair; there are no bare flags.

Retry with exponential backoff

Retry a flaky operation a bounded number of times, doubling the delay between attempts up to a cap.

```
// Retry with exponential backoff. The operation is a named function so the
// loop works on both backends; flaky succeeds on its 4th call.
fn flaky(attempt) {
  if attempt < 3 { return bad("temporary failure " + str(attempt)) }
  ok("done on attempt " + str(attempt))
}

fn backoff_ms(attempt, base, cap) { min(cap, base * pow(2, attempt)) }

fn is_ok(r) { match r { ok(v) => true, bad(e) => false } }

fn with_retry(max_attempts) {
  bind attempt = 0
  bind result = bad("never ran")
  bind going = true
  while going {
    bind result = flaky(attempt)
    if is_ok(result) {
      bind going = false
    } else {
      println("attempt ", attempt, " failed; ", result)
      bind attempt = attempt + 1
      if attempt >= max_attempts {
        bind going = false
      } else {
        bind delay = backoff_ms(attempt - 1, 10, 200)
        println("  sleeping ", delay, " ms")
        sleep(delay)
      }
    }
  }
  result
}

bind start = do {
  println("final: ", with_retry(5))
  println("final: ", with_retry(2))
}
```

```
OUTPUT
attempt 0 failed; Err(temporary failure 0)
  sleeping 10 ms
attempt 1 failed; Err(temporary failure 1)
  sleeping 20 ms
attempt 2 failed; Err(temporary failure 2)
  sleeping 40 ms
final: Ok(done on attempt 3)
attempt 0 failed; Err(temporary failure 0)
  sleeping 10 ms
attempt 1 failed; Err(temporary failure 1)
final: Err(temporary failure 1)
```

`backoff_ms` computes `min(cap, base * 2^attempt)`. The loop is a plain `while` with a `going` flag that is updated *outside* any `match` arm (`is_ok` returns the flag), which keeps it correct on both backends. The operation itself is a named function (`flaky`): Ling cannot reliably pass function values around, so a reusable retry helper takes the *name* of what to run and dispatches in one place. Add random jitter with `rand()` to avoid synchronised retries from many clients.

Token-bucket rate limiter

Allow short bursts but cap the sustained rate, with time passed in explicitly.

```
// Token-bucket rate limiter. Time is an argument, so the logic is testable.
form Bucket { tokens, last, rate, cap }

fn make_bucket(rate, cap, now) { Bucket(cap, now, rate, cap) }

// Returns [new_bucket, allowed].
fn take(b, now) {
  bind refill = min(b.cap, b.tokens + (now - b.last) * b.rate)
  if refill >= 1 {
    [Bucket(refill - 1, now, b.rate, b.cap), true]
  } else {
    [Bucket(refill, now, b.rate, b.cap), false]
  }
}

bind start = do {
  // 2 tokens per second, burst of 3.
  bind b = make_bucket(2, 3, 0)
  bind times = [0, 0.1, 0.2, 0.3, 0.6, 1.2, 1.3, 5]
  bind i = 0
  while i < len(times) {
    bind r = take(b, times[i])
    bind b = r[0]
    println("t=", times[i], "  allowed=", r[1])
    bind i = i + 1
  }
}
```

```
OUTPUT
t=0  allowed=true
t=0.1  allowed=true
t=0.2  allowed=true
t=0.3  allowed=false
t=0.6  allowed=true
t=1.2  allowed=true
t=1.3  allowed=false
t=5  allowed=true
```

The bucket refills continuously at `rate` tokens per second up to `cap`, and each request spends one. Taking `now` as a parameter makes the limiter pure and testable: the timeline above is a unit test. In a live server pass `time_now()` (seconds since start) or `epoch_now()`. Keep one bucket per client by storing buckets in an association list keyed by client id (see *A map from an association list*).

LRU cache from lists

A fixed-size cache that evicts the least recently used key.

```
// LRU cache from two parallel lists. The most recently used key is last.
form Cache { cap, keys, vals }

fn index_of(xs, x) {
  bind i = 0
  bind at = -1
  while i < len(xs) {
    if xs[i] == x { bind at = i }
    bind i = i + 1
  }
  at
}

fn remove_at(xs, at) {
  bind out = []
  bind i = 0
  while i < len(xs) {
    if i != at { bind out = list_push(out, xs[i]) }
    bind i = i + 1
  }
  out
}

fn cache_put(c, key, val) {
  bind at = index_of(c.keys, key)
  bind keys = c.keys
  bind vals = c.vals
  if at >= 0 {
    bind keys = remove_at(keys, at)
    bind vals = remove_at(vals, at)
  }
  if len(keys) >= c.cap {            // evict the least recently used (front)
    println("  evict ", keys[0])
    bind keys = remove_at(keys, 0)
    bind vals = remove_at(vals, 0)
  }
  Cache(c.cap, list_push(keys, key), list_push(vals, val))
}
```

```
// Returns [cache, value, hit]; a hit moves the key to the recent end.
fn cache_get(c, key) {
  bind at = index_of(c.keys, key)
  if at < 0 {
    bind miss = [c, "", false]
    return miss
  }
  bind val = c.vals[at]
  bind moved = cache_put(c, key, val)
  bind out = [moved, val, true]
  out
}

bind start = do {
  bind c = Cache(2, [], [])
  bind c = cache_put(c, "a", 1)
  bind c = cache_put(c, "b", 2)
  bind r = cache_get(c, "a")
  bind c = r[0]
  println("get a -> ", r[1], " hit=", r[2])
  bind c = cache_put(c, "c", 3)
  bind r = cache_get(c, "b")
  println("get b -> hit=", r[2])
  println("keys (old to new): ", c.keys)
}
```

```
OUTPUT
get a -> 1 hit=true
  evict b
get b -> hit=false
keys (old to new): [a, c]
```

Keys and values live in parallel lists ordered from least to most recently used. A read re-inserts the key at the recent end, so eviction always removes the front. Lookups are linear, which is fine up to a few hundred entries; for more, keep an index in SQLite. Note the two

helper functions: there is no `list_remove`, so deletion builds a new list. A statement that starts a line with `[` is parsed as an index into the previous expression, which is why `cache_get` binds its result list to a name before returning it.

Event bus

Decouple producers from consumers: subscribers register for a topic, publishers do not know who is listening.

```
// Event bus: subscribers register for topics by name; publish returns how many were reached.
form Bus { topics, subs }

fn subscribe(bus, topic, who) {
  Bus(list_push(bus.topics, topic), list_push(bus.subs, who))
}

// Subscribers are named; one function dispatches to the right behaviour.
fn handle(who, topic, payload) {
  if who == "logger" { return "logger saw " + topic + ": " + to_str(payload) }
  if who == "mailer" { return "mailer queued mail for " + to_str(payload) }
  "unknown subscriber " + who
}

fn publish(bus, topic, payload) {
  bind delivered = 0
  bind i = 0
  while i < len(bus.topics) {
    if bus.topics[i] == topic {
      println("  ", handle(bus.subs[i], topic, payload))
      bind delivered = delivered + 1
    }
    bind i = i + 1
  }
  delivered
}

bind start = do {
  bind bus = Bus([], [])
  bind bus = subscribe(bus, "user.created", "logger")
  bind bus = subscribe(bus, "user.created", "mailer")
  bind bus = subscribe(bus, "user.deleted", "logger")
  println("created: ", publish(bus, "user.created", "ada"))
  println("deleted: ", publish(bus, "user.deleted", "ada"))
  println("other:   ", publish(bus, "nothing", 0))
}
```

```
OUTPUT
  logger saw user.created: ada
  mailer queued mail for ada
created: 2
  logger saw user.deleted: ada
deleted: 1
other:   0
```

Topics and subscribers are parallel lists. Because functions cannot be stored in lists and called later, subscribers are *names* and a single `handle` function dispatches on the name. That also gives you one place to add logging or error handling for every delivery. Return the delivery count so tests can assert that something was listening.

Undo and redo

Two stacks around the present value give unlimited undo and redo.

```
// Undo/redo as two stacks around the present value.
form History { past, present, future }

fn pop_last(xs) {
  bind out = []
  bind i = 0
  while i < len(xs) - 1 {
    bind out = list_push(out, xs[i])
    bind i = i + 1
  }
  out
}

fn edit(h, value) { History(list_push(h.past, h.present), value, []) }

fn undo(h) {
  if len(h.past) == 0 { return h }
  History(pop_last(h.past), h.past[len(h.past) - 1], list_push(h.future, h.present))
}

fn redo(h) {
  if len(h.future) == 0 { return h }
  History(list_push(h.past, h.present), h.future[len(h.future) - 1],
pop_last(h.future))
}

bind start = do {
  bind h = History([], "", [])
  bind h = edit(h, "H")
  bind h = edit(h, "He")
  bind h = edit(h, "Hel")
  println("now:   ", h.present)
  bind h = undo(h)
  bind h = undo(h)
  println("undo2: ", h.present)
  bind h = redo(h)
  println("redo:  ", h.present)
  bind h = edit(h, "Hex")
```

```
  println("edit:  ", h.present, "  redo available: ", len(h.future) > 0)
  bind h = undo(undo(undo(undo(h))))
  println("floor: [", h.present, "]")
}
```

```
OUTPUT
now:   Hel
undo2: H
redo:  He
edit:  Hex  redo available: false
floor: []
```

`edit` pushes the present onto `past` and clears `future` (a new edit invalidates redo, as in every editor). `undo` and `redo` move the value between stacks, and both are no-ops at the ends rather than errors. For large documents store *commands* instead of whole values; the next recipe shows how.

Command pattern with inverses

Represent edits as data so they can be logged, replayed and undone.

```
// Command pattern: commands are data, and each knows its inverse.
choose Cmd { Add(n), Scale(k), Reset(old) }

fn run_cmd(state, cmd) {
  match cmd {
    Add(n) => state + n,
    Scale(k) => state * k,
    Reset(old) => 0
  }
}

fn describe(cmd) {
  match cmd {
    Add(n) => "add " + to_str(n),
    Scale(k) => "scale by " + to_str(k),
    Reset(old) => "reset"
  }
}

// The command that undoes `cmd` when `before` was the state it ran on.
fn inverse(cmd, before) {
  match cmd {
    Add(n) => Add(0 - n),
    Scale(k) => Scale(1 / k),
    Reset(old) => Add(before)
  }
}

bind start = do {
  bind state = 10
  bind cmds = [Add(5), Scale(4), Reset(0), Add(7)]
  bind undo_log = []
  bind i = 0
  while i < len(cmds) {
    bind undo_log = list_push(undo_log, inverse(cmds[i], state))
    bind state = run_cmd(state, cmds[i])
    println(describe(cmds[i]), " -> ", state)
    bind i = i + 1
```

```
  }
  bind j = len(undo_log) - 1
  while j >= 0 {
    bind state = run_cmd(state, undo_log[j])
    println("undo ", describe(undo_log[j]), " -> ", state)
    bind j = j - 1
  }
}
```

```
OUTPUT
add 5 -> 15
scale by 4 -> 60
reset -> 0
add 7 -> 7
undo add -7 -> 0
undo add 60 -> 60
undo scale by 0.25 -> 15
undo add -5 -> 10
```

`choose Cmd` declares the command variants. `run_cmd` applies one and `inverse` builds the command that reverses it, given the state it ran on (a reset needs the old value, so the log captures `before`). Dispatch uses `match` *expressions* with the variant patterns; reading fields directly off a variant (`cmd.n`) does not work. Persisting the undo log to disk gives you crash-safe journaling.

Plugin loader

Load extra `.ling` files at run time and call their functions by name.

```
// Plugin loader: every function in a plugin must be named <stem>_..., and the file may
hold only fn definitions.
fn load_plugin(path) {
  bind r = script_load(path)
  if r[0] { println("loaded ", r[2]) } else { println("refused ", path, ": ", r[1]) }
  r[0]
}

bind start = do {
  write_file("shout.ling", "fn shout_run(text) { str_upper(text) + \"!\" }\n")
  write_file("count.ling", "fn count_run(text) { len(split(text, \" \")) }\n")
  write_file("evil.ling", "fn helper(text) { text }\n")
  load_plugin("shout.ling")
  load_plugin("count.ling")
  load_plugin("evil.ling")
  println(script_functions())
  println(script_call("shout_run", ["hello plugin"]))
  println(script_call("count_run", ["one two three"]))
  println(script_call("nothing_run", []))
  remove_file("shout.ling")
  remove_file("count.ling")
  remove_file("evil.ling")
}
```

```
OUTPUT
loaded shout
loaded count
refused evil.ling: function 'helper' must start with 'evil_' (every function in
evil.ling does)
[count_run, shout_run]
[true, HELLO PLUGIN!]
[true, 3]
[false, no script function 'nothing_run']
```

`script_load` returns `[success, message, stem]`. The rules protect the host: a plugin file may contain only `fn` definitions, and every function must be named `<stem>_...` so it cannot shadow the program's own functions (`evil.ling` is refused above). `script_call` returns

`[true, value]` or `[false, message]`. Plugins run inside the host's interpreter with the host's permissions, so only load files you trust; for untrusted code use an isolate with no capabilities (see *Concurrency and Isolates*).

State machine

Model a workflow as states and events so illegal moves are impossible to express.

```
// Finite state machine: states are a choose, transitions are one function.
choose Job { Idle, Running(attempt), Failed(reason), Done }

fn label(s) {
  match s {
    Idle() => "idle",
    Running(attempt) => "running (attempt " + to_str(attempt) + ")",
    Failed(reason) => "failed: " + reason,
    Done() => "done"
  }
}

// Unknown (state, event) pairs leave the state unchanged and say so.
fn step(s, event) {
  match s {
    Idle() => if event == "start" { Running(1) } else { s },
    Running(attempt) => if event == "finish" {
      Done()
    } else {
      if event == "error" {
        if attempt >= 2 { Failed("too many errors") } else { Running(attempt + 1) }
      } else { s }
    },
    Failed(reason) => if event == "reset" { Idle() } else { s },
    Done() => s
  }
}

bind start = do {
  bind s = Idle()
  bind events = ["finish", "start", "error", "error", "reset", "start", "finish",
"start"]
  bind i = 0
  while i < len(events) {
    bind next = step(s, events[i])
    println(label(s), "  --", events[i], "-->  ", label(next))
    bind s = next
```

```
    bind i = i + 1
  }
}
```

```
OUTPUT
idle  --finish-->  idle
idle  --start-->  running (attempt 1)
running (attempt 1)  --error-->  running (attempt 2)
running (attempt 2)  --error-->  failed: too many errors
failed: too many errors  --reset-->  idle
idle  --start-->  running (attempt 1)
running (attempt 1)  --finish-->  done
done  --start-->  done
```

States are a `choose`, and `step` is a total function from (state, event) to the next state; any pair it does not know leaves the state unchanged, which is visible in the trace (`done` ignores `start`). Because the machine is a pure function you can test every transition table row without any I/O. Note the nullary variants: they are written and matched with parentheses (`Idle()`).

Schema migrations with SQLite

Upgrade a database file in place, once, whatever version it starts at.

```
// Schema migrations with SQLite: PRAGMA user_version records how far a database has
come.
fn migration_sql(version) {
  if version == 1 { return "create table users(id integer primary key, name text not
null)" }
  if version == 2 { return "alter table users add column email text default ''" }
  if version == 3 { return "create index idx_users_name on users(name)" }
  ""
}

fn current_version() { db_query("pragma user_version")[0].user_version }

fn migrate(target) {
  bind v = current_version()
  while v < target {
    bind next = v + 1
    db_exec(migration_sql(next))
    db_exec("pragma user_version = " + str(next))
    println("  applied migration ", next)
    bind v = next
  }
  v
}

bind start = do {
  remove_file("mig.db")
  db_open("mig.db")
  println("starting at version ", current_version())
  migrate(2)
  db_exec("insert into users(name, email) values (?1, ?2)", "Ada", "ada@example.com")
  println("running again is a no-op:")
  migrate(2)
  println("a later release asks for 3:")
  println("now at version ", migrate(3))
  bind rows = db_query("select id, name, email from users")
  println(len(rows), " user: ", rows[0].name, " <", rows[0].email, ">")
  remove_file("mig.db")
}
```

```
OUTPUT
starting at version 0
  applied migration 1
  applied migration 2
running again is a no-op:
a later release asks for 3:
  applied migration 3
now at version 3
1 user: Ada <ada@example.com>
```

SQLite keeps an integer in `PRAGMA user_version`. `migrate(target)` reads it, applies each missing migration in order and bumps the number after every step, so a crash leaves the database at a known version and the next start resumes. Running it again is a no-op. Wrap each step in `begin` / `commit` statements if a migration has several statements. Never edit a shipped migration; add a new one.

Password-reset tokens

Issue a one-time token that is emailed to the user, while storing only its hash.

```
// Password-reset tokens: random token to the user, only its hash stored, short expiry, constant-time check.
form Reset { user, token_hash, expires }

fn issue_token(user, now, ttl_seconds) {
  bind token = random_hex(24)
  // Returns [record_to_store, token_to_email]. The token itself is never stored.
  bind out = [Reset(user, sha256_hex(token), now + ttl_seconds), token]
  out
}

fn check_token(rec, user, presented, now) {
  if rec.user != user { return false }
  if now > rec.expires { return false }
  hex_eq(rec.token_hash, sha256_hex(presented))
}

bind start = do {
  bind issued = issue_token("ada", 1000, 900)
  bind rec = issued[0]
  bind token = issued[1]
  println("token length: ", len(token), " hex chars")
  println("stored hash length: ", len(rec.token_hash))
  println("valid now:           ", check_token(rec, "ada", token, 1100))
  println("wrong token:         ", check_token(rec, "ada", random_hex(24), 1100))
  println("wrong user:          ", check_token(rec, "bob", token, 1100))
  println("after expiry:        ", check_token(rec, "ada", token, 1901))
}
```

```
OUTPUT
token length: 48 hex chars
stored hash length: 64
valid now:           true
wrong token:         false
wrong user:          false
after expiry:        false
```

The token comes from `random_hex(24)` (192 bits from the operating system, unlike `rand()`). Only `sha256_hex(token)` is stored, so a leaked database cannot be used to reset anyone's password. Verification re-hashes the presented token and compares with `hex_eq`, a constant-time comparison, and also checks the user and the expiry. After a successful reset, delete the record so the token is single-use. Passwords themselves must use `password_hash`, never a bare hash (see *Cryptography in Practice*).

Input validation that reports everything

Validate user input, collect all problems, and return one `ok` or `bad` value.

```
// Input validation: collect every problem, not just the first.
fn valid_email(s) {
  bind at = str_find(s, "@")
  if at < 1 { return false }
  if s.contains(" ") { return false }
  bind domain = substr(s, at + 1, len(s) - at - 1)
  str_find(domain, ".") > 0 && !ends_with(domain, ".")
}

fn problems(name, email, age_text) {
  bind errs = []
  if len(trim(name)) < 2 { bind errs = list_push(errs, "name must have at least 2
characters") }
  if len(name) > 40 { bind errs = list_push(errs, "name is too long") }
  if !valid_email(email) { bind errs = list_push(errs, "email looks invalid") }
  bind age = to_number(age_text)
  if age < 13 || age > 120 { bind errs = list_push(errs, "age must be between 13 and
120") }
  errs
}

fn validate(name, email, age_text) {
  bind errs = problems(name, email, age_text)
  if len(errs) == 0 { return ok([trim(name), str_lower(trim(email)),
to_number(age_text)]) }
  bad(errs)
}

bind start = do {
  println(validate("  Ada Lovelace ", "Ada@Example.com", "36"))
  println(validate("A", "no-at-sign", "7"))
  println(validate("Grace", "grace@", "abc"))
}
```

```
OUTPUT
Ok([Ada Lovelace, ada@example.com, 36])
Err([name must have at least 2 characters, email looks invalid, age must be between 13
and 120])
Err([email looks invalid, age must be between 13 and 120])
```

`problems` returns a list of messages; an empty list means valid. `validate` converts that into `ok(cleaned)` or `bad(errors)`, so callers use a single `match`. Cleaning (trim, lower-case the email) happens only after validation passes. Always validate on the server side even if a UI validates too, and escape on output (`html_escape`) rather than trying to strip dangerous characters on input.

Secure random IDs

Generate identifiers that cannot be guessed.

```
// Secure random IDs: use random_hex (OS entropy), never rand(), for anything
guessable-sensitive.
fn new_id(prefix) { prefix + "_" + random_hex(8) }

// Human-friendly: base32 of random bytes (lower case, unpadded, no look-alike digits
0/1).
fn short_code(bytes) { base32_encode(random_hex(bytes)) }

fn all_distinct(xs) {
  bind i = 0
  bind ok_so_far = true
  while i < len(xs) {
    bind j = i + 1
    while j < len(xs) {
      if xs[i] == xs[j] { bind ok_so_far = false }
      bind j = j + 1
    }
    bind i = i + 1
  }
  ok_so_far
}

bind start = do {
  bind id = new_id("usr")
  println("shape ok: ", starts_with(id, "usr_"), " length ", len(id))
  bind code = short_code(5)
  println("short code length: ", len(code))
  bind ids = []
  bind n = 0
  while n < 200 {
    bind ids = list_push(ids, new_id("ord"))
    bind n = n + 1
  }
  println("200 ids all distinct: ", all_distinct(ids))
}
```

```
OUTPUT
shape ok: true length 20
short code length: 8
200 ids all distinct: true
```

`random_hex(n)` draws `n` bytes from the operating system's entropy source. Use it for session ids, API keys and anything an attacker could try to guess. `rand()` is a seedless pseudo-random generator for gameplay and jitter, not for secrets. 8 random bytes give 64 bits, plenty to avoid collisions among millions of rows; use 16 or more for authentication tokens. `base32_encode` over the hex gives a compact, case-insensitive code for humans to read out loud.

Pagination

Slice a list into pages and return everything a UI needs to draw the pager.

```
// Pagination: a Page record carries the slice plus everything a UI needs.
form Page { items, number, pages, total, has_prev, has_next }

fn paginate(items, number, size) {
  bind total = len(items)
  bind pages = max(1, ceil(total / size))
  bind number = clamp(number, 1, pages)
  bind first = (number - 1) * size
  bind slice = []
  bind i = first
  while i < min(total, first + size) {
    bind slice = list_push(slice, items[i])
    bind i = i + 1
  }
  Page(slice, number, pages, total, number > 1, number < pages)
}

fn show(p) {
  println("page ", p.number, "/", p.pages, " (", p.total, " items) ", p.items,
          "  prev=", p.has_prev, " next=", p.has_next)
}

bind start = do {
  bind names = ["ada", "ben", "cy", "dee", "eli", "flo", "gus"]
  show(paginate(names, 1, 3))
  show(paginate(names, 3, 3))
  show(paginate(names, 99, 3))
  show(paginate([], 1, 3))
}
```

OUTPUT

```
page 1/3 (7 items) [ada, ben, cy]  prev=false next=true
page 3/3 (7 items) [gus]  prev=true next=false
page 3/3 (7 items) [gus]  prev=true next=false
page 1/1 (0 items) []  prev=false next=false
```

`ceil(total / size)` gives the page count and `clamp` pulls an out-of-range request back to a valid page, so a bad `?page=99` shows the last page instead of an error. The same arithmetic maps to SQL as `limit size offset (page - 1) * size` when the data lives in SQLite; do the slicing in the query, not in Ling, once tables grow.

Simple templating

Fill `{{placeholders}}` in a text template, escaping values by default.

```
// Simple templating: {{name}} placeholders, values HTML-escaped by default, {{{name}}}
for raw.
fn render(template, vars) {
  bind out = template
  bind i = 0
  while i < len(vars) {
    bind key = vars[i][0]
    bind val = vars[i][1]
    bind out = str_replace(out, "{{{" + key + "}}}", to_str(val))
    bind out = str_replace(out, "{{" + key + "}}", html_escape(to_str(val)))
    bind i = i + 1
  }
  out
}

bind start = do {
  bind page = "<h1>{{title}}</h1>\n<p>Hello, {{user}}! You have {{count}} new messages.
</p>\n<div>{{{badge}}}</div>"
  bind vars = [
    ["title", "Inbox"],
    ["user", "Ada <script>alert(1)</script>"],
    ["count", 3],
    ["badge", "<b>VIP</b>"]
  ]
  println(render(page, vars))
}
```

```
OUTPUT
<h1>Inbox</h1>
<p>Hello, Ada &lt;script&gt;alert(1)&lt;/script&gt;! You have 3 new messages.</p>
<div><b>VIP</b></div>
```

Escaped output is the safe default (`html_escape`), and the triple-brace form opts out for markup you built yourself. The substitution runs the raw form first so `{{{x}}}` is not half-consumed by the `{{x}}` rule. Keep templates in files and `read_file` them; this is the same idea the web crates use at larger scale.

i18n tables for multilingual UIs

Look up user-visible strings by key and language, with English as the fallback.

```
// i18n tables: one table per language, keyed messages, fallback to English, {n} placeholders.
fn table(lang) {
  if lang == "en" { return [["greeting", "Hello, {name}!"], ["items", "You have {n} items"], ["bye", "Goodbye"]] }
  if lang == "fr" { return [["greeting", "Bonjour, {name} !"], ["items", "Vous avez {n} articles"]] }
  if lang == "ja" { return [["greeting", "こんにちは、{name}さん"], ["items", "{n}個のアイテムがあります"], ["bye", "さようなら"]] }
  if lang == "zh" { return [["greeting", "你好，{name}"], ["items", "你有{n}件物品"], ["bye", "再见"]] }
  bind none = []
  none
}

fn lookup(t, key) {
  bind i = 0
  bind found = ""
  while i < len(t) {
    if t[i][0] == key { bind found = t[i][1] }
    bind i = i + 1
  }
  found
}

fn tr(lang, key) {
  bind msg = lookup(table(lang), key)
  if msg == "" { bind msg = lookup(table("en"), key) }
  if msg == "" { bind msg = "[" + key + "]" }
  msg
}

fn fill(msg, name, n) { str_replace(str_replace(msg, "{name}", name), "{n}", to_str(n))
}

bind start = do {
  bind langs = ["en", "fr", "ja", "zh", "xx"]
  bind i = 0
```

```
  while i < len(langs) {
    bind l = langs[i]
    println(l, ": ", fill(tr(l, "greeting"), "Ada", 0), " | ", fill(tr(l, "items"), "",
3), " | ", tr(l, "bye"), " | ", tr(l, "missing"))
    bind i = i + 1
  }
}
```

```
OUTPUT
en: Hello, Ada! | You have 3 items | Goodbye | [missing]
fr: Bonjour, Ada ! | Vous avez 3 articles | Goodbye | [missing]
ja: こんにちは、Adaさん | 3個のアイテムがあります | さようなら | [missing]
zh: 你好，Ada | 你有3件物品 | 再见 | [missing]
xx: Hello, Ada! | You have 3 items | Goodbye | [missing]
```

Each language is a table of `[key, text]` pairs; the French table deliberately lacks `bye`, and the unknown language `xx` has no table at all, so both fall back to English, and an unknown key shows as `[key]` so missing translations are obvious in a screenshot. Placeholders (`{name}`, `{n}`) are replaced after lookup. Ling source and string literals are UTF-8, so Japanese and Chinese text works directly; `len` counts characters, not bytes. In a real app move each table to its own file and load it with `read_file`.

Crash reporter

Make every failure produce a diagnosis file instead of a half-finished run.

```
// Crash reporter: every stage returns ok/bad; the first bad result writes a report and stops cleanly.
fn stage_load(path) {
  if !file_exists(path) { return bad("input file missing: " + path) }
  ok(read_file(path))
}

fn stage_parse(text) {
  bind n = to_number(trim(text))
  if n == 0 && trim(text) != "0" { return bad("not a number: " + trim(text)) }
  ok(n)
}

fn stage_compute(n) {
  if n < 0 { return bad("negative input " + str(n)) }
  ok(sqrt(n))
}

fn report_crash(stage, reason, context) {
  bind body = "stage:   " + stage + "\nreason:  " + reason + "\ncontext: " + context + "\n"
  write_file("crash-report.txt", body)
  println("CRASH in ", stage, ": ", reason, " (report written)")
}

fn payload(r) { match r { ok(v) => v, bad(e) => e } }
fn is_ok(r) { match r { ok(v) => true, bad(e) => false } }

fn pipeline(path) {
  bind r1 = stage_load(path)
  if !is_ok(r1) { report_crash("load", payload(r1), path) return false }
  bind r2 = stage_parse(payload(r1))
  if !is_ok(r2) { report_crash("parse", payload(r2), path) return false }
  bind r3 = stage_compute(payload(r2))
  if !is_ok(r3) { report_crash("compute", payload(r3), path) return false }
  println("result: ", payload(r3))
  true
}
```

```
bind start = do {
  write_file("in-good.txt", "144\n")
  write_file("in-bad.txt", "twelve\n")
  pipeline("in-good.txt")
  pipeline("in-bad.txt")
  println(read_file("crash-report.txt"))
  pipeline("in-missing.txt")
  remove_file("in-good.txt")
  remove_file("in-bad.txt")
  remove_file("crash-report.txt")
}
```

```
OUTPUT
result: 12
CRASH in parse: not a number: twelve (report written)
stage:   parse
reason:  not a number: twelve
context: in-bad.txt

CRASH in load: input file missing: in-missing.txt (report written)
```

Each stage returns `ok(...)` or `bad(reason)`. The pipeline stops at the first `bad`, writes a report that names the stage, the reason and the input, and returns `false` to the caller. Ling has no exceptions; `try(closure)` exists but only works reliably on the interpreter (see *How Ling Runs*), so result values are the portable way. Add the program version and `epoch_now()` to the report, and keep the file in a known folder so users can attach it to a bug report.

Feature flags with stable rollout

Turn a feature on for a percentage of users, the same users every time.

```
// Feature flags with stable percentage rollout: the same user always lands in the same bucket.
fn bucket(flag, user) { hash_int(flag + ":" + user, 100) }

fn enabled(flag, percent, user, allow_list) {
  bind i = 0
  while i < len(allow_list) {
    if allow_list[i] == user { return true }
    bind i = i + 1
  }
  bucket(flag, user) < percent
}

bind start = do {
  bind users = ["ada", "ben", "cy", "dee", "eli", "flo", "gus", "hal", "ivy", "jo"]
  bind on = 0
  bind i = 0
  while i < len(users) {
    bind hit = enabled("new-checkout", 50, users[i], ["jo"])
    if hit { bind on = on + 1 }
    println(users[i], ": bucket ", bucket("new-checkout", users[i]), "  enabled=", hit)
    bind i = i + 1
  }
  println(on, " of ", len(users), " enabled at 50 percent (jo is on the allow list)")
  println("stable: ", enabled("new-checkout", 50, "ada", []) == enabled("new-checkout", 50, "ada", []))
}
```

```
OUTPUT
ada: bucket 10  enabled=true
ben: bucket 43  enabled=true
cy: bucket 90  enabled=false
dee: bucket 24  enabled=true
eli: bucket 54  enabled=false
flo: bucket 13  enabled=true
gus: bucket 3  enabled=true
hal: bucket 63  enabled=false
ivy: bucket 12  enabled=true
jo: bucket 93  enabled=true
7 of 10 enabled at 50 percent (jo is on the allow list)
stable: true
```

`hash_int(text, 100)` is a deterministic FNV-1a hash reduced to 0..99, so a user's bucket depends only on the flag name and user id; raising the percentage only ever adds users. Salting with the flag name makes different flags roll out to different groups. An allow list overrides the bucket for testers. The hash is not cryptographic and must not be used for anything adversarial.

A map from an association list

Ling has no dictionary type; this is the idiomatic substitute.

```
// A map from an association list: [[key, value], ...] with get/set/has/delete/keys.
fn map_index(m, key) {
  bind i = 0
  bind at = -1
  while i < len(m) {
    if m[i][0] == key { bind at = i }
    bind i = i + 1
  }
  at
}

fn map_get(m, key, fallback) {
  bind at = map_index(m, key)
  if at < 0 { return fallback }
  m[at][1]
}

fn map_has(m, key) { map_index(m, key) >= 0 }

// Returns a new map; replaces the value if the key exists, appends otherwise.
fn map_set(m, key, value) {
  bind at = map_index(m, key)
  if at < 0 { return list_push(m, [key, value]) }
  list_set(m, at, [key, value])
}

fn map_delete(m, key) {
  bind out = []
  bind i = 0
  while i < len(m) {
    if m[i][0] != key { bind out = list_push(out, m[i]) }
    bind i = i + 1
  }
  out
}

fn map_keys(m) {
  bind out = []
```

```
  bind i = 0
  while i < len(m) {
    bind out = list_push(out, m[i][0])
    bind i = i + 1
  }
  out
}

bind start = do {
  bind m = []
  bind m = map_set(m, "lang", "ling")
  bind m = map_set(m, "year", 2030)
  bind m = map_set(m, "lang", "Ling")
  println("lang: ", map_get(m, "lang", "?"))
  println("size: ", len(m), "  keys: ", map_keys(m))
  println("has year: ", map_has(m, "year"), "  has none: ", map_has(m, "none"))
  bind m = map_delete(m, "year")
  println("after delete: ", m)
  println("missing: ", map_get(m, "year", "default"))
}
```

```
OUTPUT
lang: Ling
size: 2  keys: [lang, year]
has year: true  has none: false
after delete: [[lang, Ling]]
missing: default
```

A map is a list of `[key, value]` pairs. `map_set` returns a new list (replace if present, append otherwise), `map_delete` rebuilds without the key, and every operation is linear. That is perfectly good for dozens or a few hundred keys and keeps data printable and serialisable. For thousands of keys use a SQLite table with a primary key. Do not compare maps with `==`: lists never compare equal in Ling, so compare keys and values explicitly.

Word frequencies and top-N

Count occurrences and report the most frequent without a sort builtin.

```
// Word frequencies and top-N, using the association-list map and a selection pass (no sort builtin needed).
fn map_index(m, key) {
  bind i = 0
  bind at = -1
  while i < len(m) {
    if m[i][0] == key { bind at = i }
    bind i = i + 1
  }
  at
}

fn count_words(text) {
  bind words = split(str_lower(text), " ")
  bind counts = []
  bind i = 0
  while i < len(words) {
    bind w = trim(words[i])
    if len(w) > 0 {
      bind at = map_index(counts, w)
      if at < 0 {
        bind counts = list_push(counts, [w, 1])
      } else {
        bind counts = list_set(counts, at, [w, counts[at][1] + 1])
      }
    }
    bind i = i + 1
  }
  counts
}

// Highest count first; ties keep first-seen order.
fn top_n(counts, n) {
  bind remaining = counts
  bind out = []
  bind picked = 0
  while picked < n && len(remaining) > 0 {
    bind best = 0
```

```
    bind i = 1
    while i < len(remaining) {
      if remaining[i][1] > remaining[best][1] { bind best = i }
      bind i = i + 1
    }
    bind out = list_push(out, remaining[best])
    bind rest = []
    bind j = 0
    while j < len(remaining) {
      if j != best { bind rest = list_push(rest, remaining[j]) }
      bind j = j + 1
    }
    bind remaining = rest
    bind picked = picked + 1
  }
  out
}

bind start = do {
  bind text = "the quick brown fox jumps over the lazy dog the fox barks and the dog
runs"
  bind top = top_n(count_words(text), 3)
  bind i = 0
  while i < len(top) {
    println(top[i][0], " x", top[i][1])
    bind i = i + 1
  }
}
```

```
OUTPUT
the x4
fox x2
dog x2
```

Counting uses the association-list pattern. `top_n` does repeated selection: find the best remaining entry, move it to the output, and rebuild the rest. For small `n` that is O(n * k) and avoids writing a full sort. Use `merge_sort` below when you need the whole list ordered.

FIFO queue with a moving head

Pop from the front of a list without copying the list on every pop.

```
// FIFO queue with a moving head: push is O(1) amortised, pop never copies the whole list.
form Queue { items, head }

fn q_new() { Queue([], 0) }
fn q_push(q, x) { Queue(list_push(q.items, x), q.head) }
fn q_len(q) { len(q.items) - q.head }
fn q_peek(q) { q.items[q.head] }

// Returns [queue, item]. Compacts when more than half the list is dead space.
fn q_pop(q) {
  bind item = q.items[q.head]
  bind head = q.head + 1
  bind items = q.items
  if head > 8 && head * 2 > len(items) {
    bind fresh = []
    bind i = head
    while i < len(items) {
      bind fresh = list_push(fresh, items[i])
      bind i = i + 1
    }
    bind items = fresh
    bind head = 0
  }
  bind out = [Queue(items, head), item]
  out
}

bind start = do {
  bind q = q_new()
  bind n = 0
  while n < 20 {
    bind q = q_push(q, "job" + str(n))
    bind n = n + 1
  }
  bind served = 0
  bind last = ""
  while q_len(q) > 5 {
```

```
    bind r = q_pop(q)
    bind q = r[0]
    bind last = r[1]
    bind served = served + 1
  }
  println("served ", served, ", last was ", last)
  println("waiting: ", q_len(q), ", next is ", q_peek(q))
  println("internal list length after compaction: ", len(q.items), ", head ", q.head)
}
```

```
OUTPUT
served 15, last was job14
waiting: 5, next is job15
internal list length after compaction: 9, head 4
```

Popping by rebuilding the list costs O(n) each time. Here a `head` index advances instead, and the dead prefix is dropped only when it exceeds half the list (compaction), which makes pops cheap on average. The pop function returns `[queue, item]` because there is no in-place mutation.

Atomic save with backup

Never leave a half-written file behind: write to a temporary file, then rename.

```
// Atomic save with a backup: write a temp file, keep the old version as .bak, then
rename into place.
fn save_atomic(path, text) {
  bind tmp = path + ".tmp"
  write_file(tmp, text)
  if file_exists(path) { copy_file(path, path + ".bak") }
  rename_file(tmp, path)
  file_exists(path) && !file_exists(tmp)
}

bind start = do {
  println("first save:  ", save_atomic("doc.txt", "version 1\n"))
  println("second save: ", save_atomic("doc.txt", "version 2\n"))
  println("current: ", trim(read_file("doc.txt")))
  println("backup:  ", trim(read_file("doc.txt.bak")))
  println("temp left behind: ", file_exists("doc.txt.tmp"))
  remove_file("doc.txt")
  remove_file("doc.txt.bak")
}
```

```
OUTPUT
first save:  true
second save: true
current: version 2
backup:  version 1
temp left behind: false
```

A rename within one folder replaces the target in one step, so readers see either the old or the new file, never a torn one. The previous version is copied to `.bak` first. If the program crashes before the rename the original is untouched, and the stale `.tmp` can be removed on the next start.

CSV with quoted fields

Parse and write CSV lines whose fields contain commas and quotes.

```
// CSV: read quoted fields (commas and doubled quotes inside quotes) one character at a time, and write them back.
// Characters are taken with substr because iterating a string with `for` is not portable across backends.
fn parse_line(line) {
  bind fields = []
  bind cur = ""
  bind quoted = false
  bind i = 0
  while i < len(line) {
    bind c = substr(line, i, 1)
    if quoted {
      if c == "\"" {
        if i + 1 < len(line) && substr(line, i + 1, 1) == "\"" {
          bind cur = cur + "\""
          bind i = i + 1
        } else {
          bind quoted = false
        }
      } else {
        bind cur = cur + c
      }
    } else {
      if c == "\"" {
        bind quoted = true
      } else {
        if c == "," {
          bind fields = list_push(fields, cur)
          bind cur = ""
        } else {
          bind cur = cur + c
        }
      }
    }
    bind i = i + 1
  }
  list_push(fields, cur)
}
```

```
fn quote(field) {
  if field.contains(",") || field.contains("\"") || field.contains("\n") {
    return "\"" + str_replace(field, "\"", "\"\"") + "\""
  }
  field
}

fn format_line(fields) {
  bind out = []
  bind i = 0
  while i < len(fields) {
    bind out = list_push(out, quote(fields[i]))
    bind i = i + 1
  }
  list_join(out, ",")
}

bind start = do {
  bind line = "42,\"Lovelace, Ada\",\"says \"\"hi\"\"\",,end"
  bind fields = parse_line(line)
  println(len(fields), " fields: ", fields)
  bind back = format_line(fields)
  println(back)
  println("round trip exact: ", back == line)
}
```

```
OUTPUT
5 fields: [42, Lovelace, Ada, says "hi", , end]
42,"Lovelace, Ada","says ""hi""",,end
round trip exact: true
```

The parser is a small state machine over characters (`substr(line, i, 1)`), with a `quoted` flag and the standard doubled-quote escape. Characters are read with `substr` and an index because `for c in "text"` yields nothing on the default backend. `format_line` quotes only the fields that need it, so the round trip reproduces the input exactly (printed above as `true`).

Sorting and binary search

Order numbers in O(n log n) and find values in O(log n).

```
// Sorting numbers and binary search. There is no sort builtin, and text cannot be ordered with < on every
// backend, so sort numbers (or map text to numbers first). Merge sort keeps it O(n log n).
fn merge(a, b) {
  bind out = []
  bind i = 0
  bind j = 0
  while i < len(a) || j < len(b) {
    bind take_a = false
    if i < len(a) {
      if j >= len(b) { bind take_a = true } else { if a[i] <= b[j] { bind take_a = true } }
    }
    if take_a {
      bind out = list_push(out, a[i])
      bind i = i + 1
    } else {
      bind out = list_push(out, b[j])
      bind j = j + 1
    }
  }
  out
}

fn slice(xs, from, to) {
  bind out = []
  bind i = from
  while i < to {
    bind out = list_push(out, xs[i])
    bind i = i + 1
  }
  out
}

fn merge_sort(xs) {
  if len(xs) <= 1 { return xs }
  bind mid = floor(len(xs) / 2)
```

```
  merge(merge_sort(slice(xs, 0, mid)), merge_sort(slice(xs, mid, len(xs))))
}

// Index of x in the sorted list, or -1.
fn binary_search(xs, x) {
  bind lo = 0
  bind hi = len(xs) - 1
  while lo <= hi {
    bind mid = floor((lo + hi) / 2)
    if xs[mid] == x { return mid }
    if xs[mid] < x { bind lo = mid + 1 } else { bind hi = mid - 1 }
  }
  return -1
}

bind start = do {
  bind data = [41, 7, 19, 3, 88, 7, 23, 56, 1]
  bind sorted = merge_sort(data)
  println("sorted:   ", sorted)
  println("original: ", data)
  println("find 23:  index ", binary_search(sorted, 23))
  println("find 24:  index ", binary_search(sorted, 24))
}
```

```
OUTPUT
sorted:   [1, 3, 7, 7, 19, 23, 41, 56, 88]
original: [41, 7, 19, 3, 88, 7, 23, 56, 1]
find 23:  index 5
find 24:  index -1
```

Ling has neither a sort builtin nor a reliable `<` on text (it is `false` on the JIT and an error on the interpreter), so sort numbers, or map text to numbers first (for example by length or by a numeric code). `merge_sort` returns a new list and leaves the input alone. The `return -1` at the end of `binary_search` is written with `return`: a bare `-1` on its own line after a `}` is parsed as subtraction from the loop.

Semantic version comparison

Compare dotted version strings numerically, so `1.10` is newer than `1.2`.

```
// Semantic version comparison: returns -1, 0 or 1. Missing parts count as 0.
fn part(parts, i) {
  if i < len(parts) { return to_number(parts[i]) }
  0
}

fn compare_versions(a, b) {
  bind pa = split(a, ".")
  bind pb = split(b, ".")
  bind n = max(len(pa), len(pb))
  bind i = 0
  while i < n {
    bind x = part(pa, i)
    bind y = part(pb, i)
    if x < y { return -1 }
    if x > y { return 1 }
    bind i = i + 1
  }
  0
}

fn at_least(version, minimum) { compare_versions(version, minimum) >= 0 }

bind start = do {
  println("1.2.3 vs 1.10.0: ", compare_versions("1.2.3", "1.10.0"))
  println("2.0 vs 2.0.0:    ", compare_versions("2.0", "2.0.0"))
  println("3.1.4 vs 3.1.3:  ", compare_versions("3.1.4", "3.1.3"))
  println("plugin api ok:   ", at_least("2030.1.53", "2030.1.19"))
}
```

```
OUTPUT
1.2.3 vs 1.10.0: -1
2.0 vs 2.0.0:    0
3.1.4 vs 3.1.3:  1
plugin api ok:   true
```

Compare component by component as numbers, padding the shorter version with zeros. Plain text comparison would put `1.10.0` before `1.2.3`. Use it to gate plugins and file formats on a minimum version.

Latency metrics and percentiles

Summarise timing samples with mean, max and nearest-rank percentiles.

```
// Latency metrics: count, mean, max and a nearest-rank percentile over recorded
samples.
fn insertion_sort(xs) {
  bind out = []
  bind i = 0
  while i < len(xs) {
    bind x = xs[i]
    bind at = len(out)
    bind k = 0
    while k < len(out) {
      if out[k] > x && at == len(out) { bind at = k }
      bind k = k + 1
    }
    bind next = []
    bind m = 0
    while m < at {
      bind next = list_push(next, out[m])
      bind m = m + 1
    }
    bind next = list_push(next, x)
    while m < len(out) {
      bind next = list_push(next, out[m])
      bind m = m + 1
    }
    bind out = next
    bind i = i + 1
  }
  out
}

fn percentile(sorted, p) {
  if len(sorted) == 0 { return 0 }
  bind rank = max(1, ceil(p / 100 * len(sorted)))
  sorted[rank - 1]
}

fn mean(xs) {
  bind total = 0
```

```
  bind i = 0
  while i < len(xs) {
    bind total = total + xs[i]
    bind i = i + 1
  }
  total / max(1, len(xs))
}

bind start = do {
  bind ms = [120, 95, 101, 480, 99, 110, 105, 98, 2300, 102]
  bind sorted = insertion_sort(ms)
  println("count: ", len(ms), "  mean: ", mean(ms), "  max: ", sorted[len(sorted) - 1])
  println("p50: ", percentile(sorted, 50), "  p90: ", percentile(sorted, 90), "  p99:
", percentile(sorted, 99))
}
```

```
OUTPUT
count: 10  mean: 361  max: 2300
p50: 102  p90: 480  p99: 2300
```

Sort a copy, then the p-th percentile is the element at rank `ceil(p/100 * n)`. The sample here shows why you look at percentiles rather than the mean: the mean is 361 ms while the median is 102 ms, dragged up by two outliers. Record samples with `time_now()` differences around the code you care about.

Circuit breaker

Stop hammering a failing dependency, then probe carefully to see when it recovers.

```
// Circuit breaker: closed -> open after N failures, open -> half-open after a cool-
down, half-open -> closed on success.
form Breaker { state, failures, opened_at, threshold, cooldown }

fn make_breaker(threshold, cooldown) { Breaker("closed", 0, 0, threshold, cooldown) }

// May a call go through at time `now`? Returns [breaker, allowed].
fn allow(b, now) {
  if b.state == "open" {
    if now - b.opened_at >= b.cooldown {
      bind probe = [Breaker("half-open", b.failures, b.opened_at, b.threshold,
b.cooldown), true]
      return probe
    }
    bind refused = [b, false]
    return refused
  }
  bind passed = [b, true]
  passed
}

fn record(b, success, now) {
  if success { return Breaker("closed", 0, 0, b.threshold, b.cooldown) }
  bind failures = b.failures + 1
  if b.state == "half-open" || failures >= b.threshold {
    return Breaker("open", failures, now, b.threshold, b.cooldown)
  }
  Breaker(b.state, failures, b.opened_at, b.threshold, b.cooldown)
}

bind start = do {
  bind b = make_breaker(3, 10)
  // [time, did the downstream call succeed if we make it?]
  bind script = [[1, false], [2, false], [3, false], [4, true], [8, true], [14, false],
[25, true], [26, true]]
  bind i = 0
  while i < len(script) {
    bind now = script[i][0]
```

```
    bind r = allow(b, now)
    bind b = r[0]
    if r[1] {
      bind b = record(b, script[i][1], now)
      println("t=", now, " call made, success=", script[i][1], " -> ", b.state)
    } else {
      println("t=", now, " refused (breaker ", b.state, ")")
    }
    bind i = i + 1
  }
}
```

```
OUTPUT
t=1 call made, success=false -> closed
t=2 call made, success=false -> closed
t=3 call made, success=false -> open
t=4 refused (breaker open)
t=8 refused (breaker open)
t=14 call made, success=false -> open
t=25 call made, success=true -> closed
t=26 call made, success=true -> closed
```

After `threshold` consecutive failures the breaker opens and refuses calls for `cooldown` seconds. The first call after the cool-down is a half-open probe: success closes the breaker, failure re-opens it immediately (see `t=14` and `t=25` in the trace). As with the rate limiter, time is a parameter. Combine it with the retry recipe so retries stop when the breaker opens.

Cache with expiry

Remember values for a limited time without background threads.

```
// A cache with per-entry expiry. The clock is a parameter, so expiry can be tested without sleeping.
form Entry { key, value, expires }

fn find(entries, key, now) {
  bind i = 0
  bind found = -1
  while i < len(entries) {
    if entries[i].key == key && entries[i].expires > now { bind found = i }
    bind i = i + 1
  }
  found
}

fn cache_set(entries, key, value, now, ttl) {
  // Drop anything expired or being replaced, then append.
  bind keep = []
  bind i = 0
  while i < len(entries) {
    if entries[i].expires > now && entries[i].key != key { bind keep = list_push(keep, entries[i]) }
    bind i = i + 1
  }
  list_push(keep, Entry(key, value, now + ttl))
}

fn cache_get(entries, key, now, fallback) {
  bind at = find(entries, key, now)
  if at < 0 { return fallback }
  entries[at].value
}

bind start = do {
  bind c = []
  bind c = cache_set(c, "token", "abc123", 100, 30)
  bind c = cache_set(c, "profile", "Ada", 100, 300)
  println("t=110 token:   ", cache_get(c, "token", 110, "(miss)"))
  println("t=131 token:   ", cache_get(c, "token", 131, "(miss)"))
```

```
  println("t=131 profile: ", cache_get(c, "profile", 131, "(miss)"))
  bind c = cache_set(c, "token", "def456", 140, 30)
  println("entries kept after refresh at t=140: ", len(c))
}
```

```
OUTPUT
t=110 token:   abc123
t=131 token:   (miss)
t=131 profile: Ada
entries kept after refresh at t=140: 2
```

Each entry carries its expiry time, and a lookup ignores entries that have expired, so expiry needs no timer. Writes sweep expired entries out, which bounds memory. Pass `now` in so the behaviour is testable: the trace above checks the 30-second token at 110 (hit) and 131 (miss) without sleeping.

Structured JSON log lines

Emit one JSON object per line so a log shipper or `grep` can parse your logs.

```
// Structured logging: one JSON object per line, safe against quotes, backslashes and
newlines in messages.
// Numbers are written bare, everything else is escaped text. A field is [key, value,
"n"] for a number.
fn json_field(f) {
  if len(f) == 3 {
    if f[2] == "n" { return "\"" + f[0] + "\":" + to_str(f[1]) }
  }
  "\"" + f[0] + "\":\"" + json_escape(to_str(f[1])) + "\""
}

fn json_line(fields) {
  bind parts = []
  bind i = 0
  while i < len(fields) {
    bind parts = list_push(parts, json_field(fields[i]))
    bind i = i + 1
  }
  "{" + list_join(parts, ",") + "}"
}

bind start = do {
  println(json_line([["level", "info"], ["msg", "user said \"hi\""], ["ms", 42, "n"]]))
  println(json_line([["level", "error"], ["msg", "line one\nline two\tTabbed"],
["path", "C:\\temp\\x"]]))
}
```

```
OUTPUT
{"level":"info","msg":"user said \"hi\"","ms":42}
{"level":"error","msg":"line one\nline two\tTabbed","path":"C:\\temp\\x"}
```

`json_escape` handles quotes, backslashes, newlines and tabs so a hostile message cannot break the line or inject fields. Fields are `[key, value]`, with a third element marking numbers that should be written bare. The type check is written as two nested `if`s on purpose: `if len(f) == 3 && f[2] == "n"` reads `f[2]` even when `len(f)` is 2 on the interpreter (`&&` does not short-circuit there) and fails with an index error.

Settings persistence with versioned upgrade

Load user settings over built-in defaults, upgrade old files, and save them back.

```
// Settings persistence: defaults, saved overrides, and a schema version so old files are upgraded.
fn defaults() { [["version", "2"], ["theme", "dark"], ["font_size", "14"], ["autosave", "true"]] }

fn get(pairs, key, fallback) {
  bind i = 0
  bind found = fallback
  while i < len(pairs) {
    if pairs[i][0] == key { bind found = pairs[i][1] }
    bind i = i + 1
  }
  found
}

fn parse(text) {
  bind out = []
  bind lines = split(text, "\n")
  bind i = 0
  while i < len(lines) {
    bind at = str_find(lines[i], "=")
    if at > 0 { bind out = list_push(out, [substr(lines[i], 0, at), substr(lines[i], at + 1, len(lines[i]) - at - 1)]) }
    bind i = i + 1
  }
  out
}

// Defaults first, then whatever the file overrides; version-1 files used "size" for the font.
fn load(path) {
  bind saved = []
  if file_exists(path) { bind saved = parse(read_file(path)) }
  bind merged = []
  bind d = defaults()
  bind i = 0
  while i < len(d) {
    bind merged = list_push(merged, [d[i][0], get(saved, d[i][0], d[i][1])])
```

```
    bind i = i + 1
  }
  if get(saved, "version", "1") == "1" {
    bind merged = list_set(merged, 2, ["font_size", get(saved, "size", "14")])
    bind merged = list_set(merged, 0, ["version", "2"])
  }
  merged
}

fn save(path, settings) {
  bind lines = []
  bind i = 0
  while i < len(settings) {
    bind lines = list_push(lines, settings[i][0] + "=" + settings[i][1])
    bind i = i + 1
  }
  write_file(path, list_join(lines, "\n") + "\n")
}

bind start = do {
  println("no file:      ", load("settings.ini"))
  write_file("settings.ini", "version=1\ntheme=light\nsize=18\n")
  bind s = load("settings.ini")
  println("migrated v1:  ", s)
  save("settings.ini", s)
  println("saved as:     ", split(trim(read_file("settings.ini")), "\n"))
  println("reload equal: ", load("settings.ini") == s, "  (lists never compare equal
with ==)")
  println("font size:    ", get(load("settings.ini"), "font_size", "?"))
  remove_file("settings.ini")
}
```

```
OUTPUT
no file:      [[version, 2], [theme, dark], [font_size, 14], [autosave, true]]
migrated v1:  [[version, 2], [theme, light], [font_size, 18], [autosave, true]]
saved as:     [version=2, theme=light, font_size=18, autosave=true]
reload equal: false  (lists never compare equal with ==)
font size:    18
```

Defaults come first and the file overrides them key by key, so adding a new setting in a later release needs no migration. A `version` field handles renames: the version 1 file used `size`, which is mapped to `font_size` on load. Saving always writes the current version. The last line shows the `==` trap again: comparing two identical lists prints `false`, so compare the pieces you care about.

Chapter summary

- Ling's building blocks (records, lists, `while`, `match`, result values) are enough to build every pattern here; the cost is that you write the helpers yourself.
- Pass state and time explicitly; return new values instead of mutating.
- Prefer `ok` / `bad` results over relying on how the JIT and interpreter happen to fail.
- Use `random_hex`, `sha256_hex`, `hex_eq` and `password_hash` for anything security-related, never `rand()` or hand-rolled hashing.
- When a recipe starts to strain (linear lookups, large lists), move the data to SQLite with `db_*`.

CHAPTER 13

Music Recipes: A DAW in Ling

LingStationX is a full digital audio workstation written in Ling: arranger, piano roll, mixer, node editor, performance launcher, file browser, dialogs and every instrument panel. It is about 26,000 lines of Ling in 60 modules. This chapter first shows how such a program is organised, then gives eleven short recipes taken or adapted from it. Each one stands alone.

All snippets marked `ling` were run with `ling run --interp` and the output shown is real. Snippets marked `ling-gui` open a window or need the audio engine, and were checked against the builtin signatures in the Ling runtime; the piano-roll drawing recipe was also run and its screenshot is reproduced.

Architecture of a DAW in Ling

LingStationX is split into three layers.

1. **The engine** is a native Rust crate, `ling-daw`, that owns the project, the audio device, the instruments and the effects. Ling reaches it through `daw_*` builtins: `daw_track_add`, `daw_clip_add_midi`, `daw_note_add`, `daw_play`, `daw_bounce`, `daw_undo_push` and a few hundred more. The audio thread never runs Ling code.
2. **The UI** is plain Ling. It redraws the whole window every frame with `px_*` pixel builtins (`px_rect`, `px_text`, `px_line`, and `px_notes`, which draws all the notes of a clip in one native call). There is no widget tree: a widget is a function that draws itself and returns what the user did this frame.
3. **The content** is data on disk: instrument patches (`instruments/*.ling`), project templates, effect presets, and *scripts* (`scripts/*.ling`) that users edit while the program runs.

The module layout of the `ling/` folder follows the screens:

Module	Lines	Role
`main.ling`	614	Entry point: opens the window, loads fonts, runs the frame loop
`core.ling`	685	Per-frame input (`u_pressed`, `u_hot`), clipping, the widget kit (buttons, sliders, combos, text fields)
`ds.ling`	320	The design system: palette, outlined panels, tile lettering
`actions.ling`	1770	`do_action(id)`: every menu item, key and button is a string action id

Module	Lines	Role
`arranger.ling`	842	Timeline: ruler, track tiles, clips, loop region, tools, snapping
`roll.ling`	1744	Piano roll: grid, notes, velocity and expression lanes, copy and paste
`mixer.ling`, `fxpanel.ling`	298, 1117	Channel strips, meters, effect racks
`nodes.ling`	931	Node editor for signal routing
`automation.ling`, `perf.ling`	288, 566	Automation lanes, clip launcher
`scripts.ling`	215	Loads user scripts with `script_load` and `script_call`
`growl.ling`, `prism.ling`	926, 722	Instrument panels for two built-in synths

State without globals

A Ling function sees the constants defined in `main.ling` but not its caller's variables, and cannot change them. LingStationX therefore keeps all shared state in the native store: `st_set(key, value)`, `st_get(key, default)`, and list helpers such as `st_lpush`. Keys are strings with a module prefix (`rsel` for piano-roll selection, `az` for arranger zoom). Reads are cheap and the store survives across frames, so a widget can remember that it is being dragged simply by writing `st_set("active", "left_resize")`.

The frame loop

Every frame does the same five things, in this order:

1. read the mouse and keyboard once into the store (`u_begin`);
2. draw all panels, which also handle clicks against the rectangles they just drew;
3. dispatch any pending action string through `do_action`;
4. push UI changes into the engine (`engine_sync`) and run the autosave timer;
5. `present()`.

Because input is read once and drawing is the hit test, there is no event queue and no widget lifetime to manage. The cost is that every panel must be cheap to draw; hot paths such as the notes of a clip are drawn by one native call (`px_notes`) and cached by project revision (`daw_rev`).

The instrument plugin format

An instrument is a text file. The first lines name it, then come `key=value` pairs, all in the range 0 to 1:

```
OUTPUT
# Ling Synth X patch - edit freely; load from the Instrument tab
name = "Init Saw"
type = synth
wave=0.660 sub=1.000 add=0.000 fm=0.000 cutoff=0.600 reso=0.200 filt_env=0.400
attack=0.050 decay=0.289 sustain=0.700 release=0.250 lfo_rate=0.300
```

`type` selects the engine (`synth`, `drums`, `chip`, `cinema`, `growl`, `prism`, `voice`). More than 900 patch files ship in `instruments/`, from General MIDI sounds to drum kits and synth presets. A drum patch uses dotted keys (`kick.tune`, `snare.decay`). Loading a patch is `daw_track_add(kind, path)`; saving is `daw_inst_save`. The format is deliberately one line per group so that diffs and hand edits stay readable.

For sounds the built-in engines cannot make, an instrument can be a **script**: a `.ling` file whose `<name>_dsp()` function returns a small DSP program as a string (see "An instrument written in LingDSP, rendered and measured"). The host compiles it with `daw_script_check` and adds a track with `daw_script_inst_add`. Saving the file reloads every running instance.

Arranger, piano roll and mixer patterns

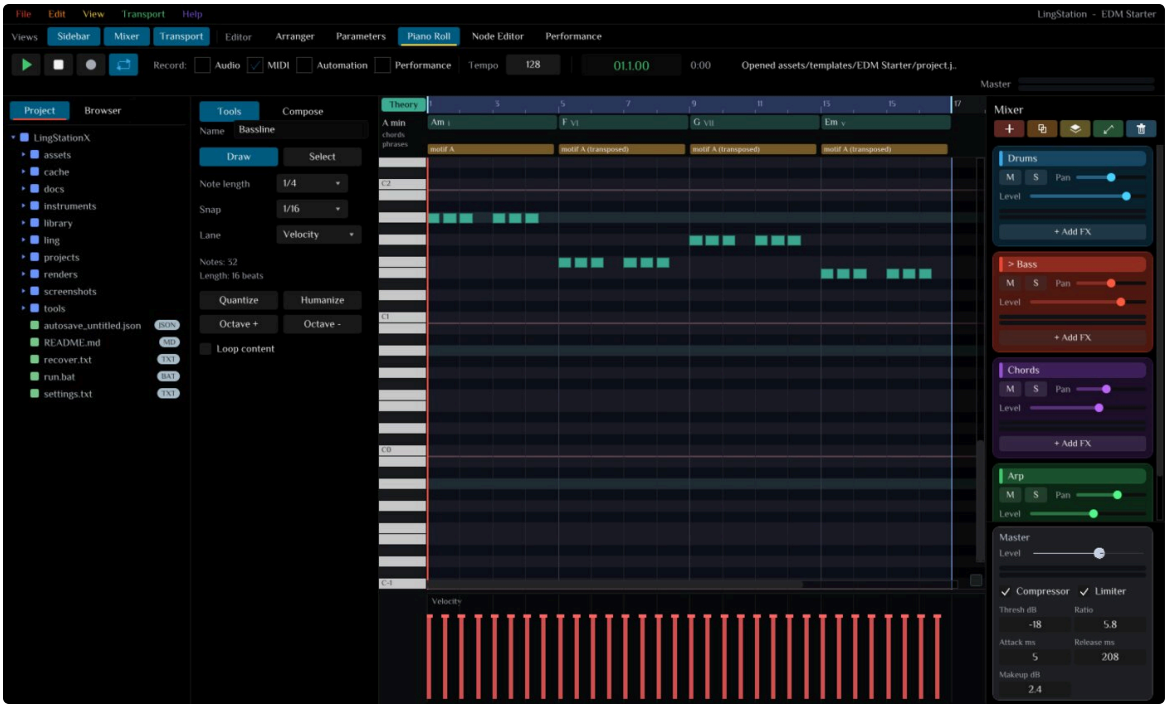


Figure 10.1. The piano roll. The grid, the notes, the keyboard strip and the velocity bars below are all drawn by `roll.ling` with `px_*` calls.

- **One coordinate system.** Time is in *beats* everywhere, never seconds. The arranger and roll convert beats to pixels with `x = gx + (beat - scroll) * beat_width`, and back for hit tests. Seconds appear only for tempo changes and audio.
- **Snap is a function.** `snap_to(v)` rounds to the current grid step, and the "cell" snap adapts to zoom so that a 16th grid appears only when the screen can show it (see "Tempo grid").
- **Edit, then checkpoint.** The engine owns the undo stack. The UI calls `daw_undo_push()` before each user edit and `daw_undo()` or `daw_redo()` on Ctrl+Z and Ctrl+Y.
- **Notes are flat lists.** `daw_clip_notes(t, c)` returns seven numbers per note: start, length, key, velocity, pan, cutoff, resonance. Scripts rewrite a clip by building a new flat list and calling `daw_clip_set_notes`.
- **Mixer strips are colour-coded rows.** Each track draws mute, solo, pan, level and a meter. The meter maps linear amplitude to a -60 to 0 dB scale (see "Mixer maths").

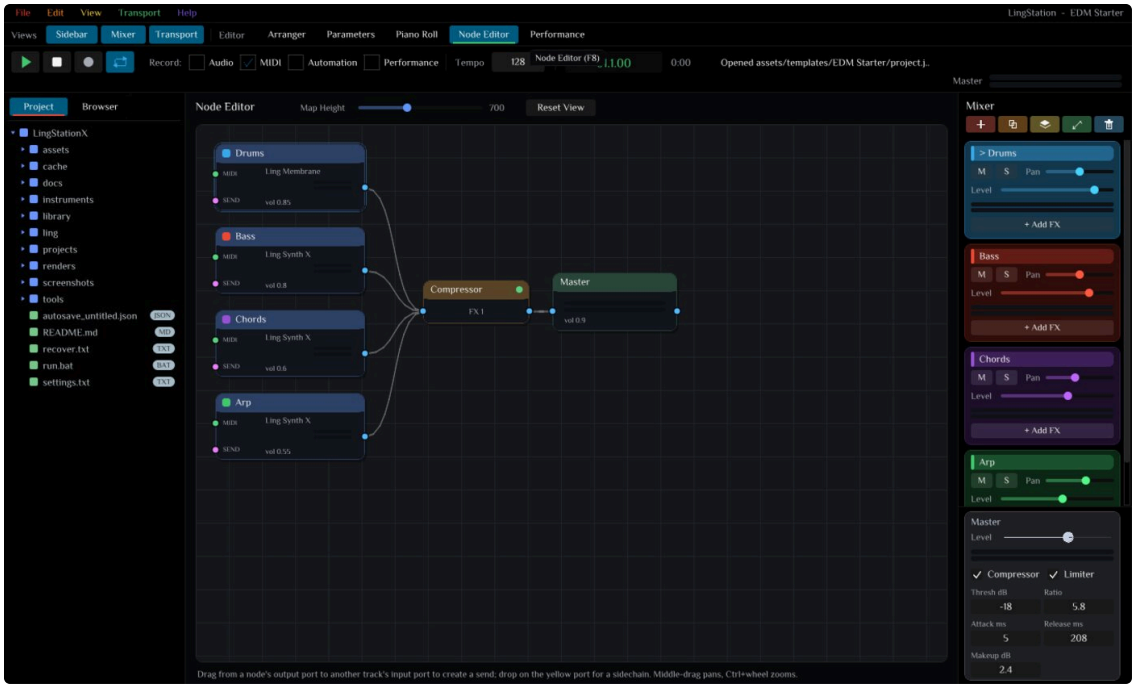


Figure 10.2. The node editor, a second view of the same project: each track is a node, and a cable is a send or a sidechain feeding another track.

- **Tests drive the real UI.** Command-line flags (`--shot N`, `--tab T`, `--do "frame,action"`, `--drive "frame,x,y,down"`) run the application, inject scripted mouse input, take a screenshot and quit. The screenshots in this book were produced this way.

Note number to frequency and name

MIDI notes are integers, instruments want hertz, and the UI wants names such as `C#4`. Equal temperament makes all three one line each. LingStationX's theory panel uses the same pitch-class table to label the piano roll.

```
fn midi_hz(k) { return 440.0 * pow(2.0, (k - 69.0) / 12.0) }
fn pc_name(k) { return list_get(["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A",
"A#", "B"], k % 12.0) }
fn note_name(k) { return pc_name(k) + num_str(floor(k / 12.0) - 1.0) }
fn hz_midi(f) { return 69.0 + 12.0 * log(f / 440.0) / log(2.0) }
bind start = do {
  bind ks = [21.0, 60.0, 69.0, 72.0, 108.0]
  bind i = 0.0
  while i < len(ks) {
    bind k = list_get(ks, i)
    println(note_name(k), "  key ", k, "  ", round(midi_hz(k) * 100.0) / 100.0, " Hz")
    bind i = i + 1.0
  }
  println("261.63 Hz is key ", round(hz_midi(261.63)))
}
```

```
OUTPUT
A0  key 21  27.5 Hz
C4  key 60  261.63 Hz
A4  key 69  440 Hz
C5  key 72  523.25 Hz
C8  key 108  4186.01 Hz
261.63 Hz is key 60
```

How it works. Key 69 is A4 at 440 Hz and every semitone multiplies the frequency by 2^(1/12). The octave number is `floor(key / 12) - 1`, so key 60 is C4 ("middle C"). `hz_midi` inverts the formula with a logarithm.

Variations. For microtonal tuning pass a fractional key (`60.5`). To tune to A=432 Hz change the constant 440. Keep the table in `list_get([...], key % 12)`; a lookup table is faster than building the name by string logic.

Tempo grid: beats, bars and seconds

The arranger shows bars and beats; the audio engine needs seconds. With a tempo map in the project, the conversion is a sum over segments. This recipe also shows the zoom-adaptive snap step used by the arranger.

```
# tempo map: [beat, bpm] points, tempo holds until the next point
fn tm_secs(points, beat) {
  bind secs = 0.0
  bind i = 0
  while i < len(points) {
    bind p = list_get(points, i)
    bind b0 = list_get(p, 0)
    bind bpm = list_get(p, 1)
    bind b1 = beat
    if i + 1 < len(points) {
      bind nxt = list_get(list_get(points, i + 1), 0)
      if nxt < beat { bind b1 = nxt }
    }
    if b1 > b0 { bind secs = secs + (b1 - b0) * 60.0 / bpm }
    bind i = i + 1
  }
  return secs
}
fn bar_beat(beat) {
  bind bar = floor(beat / 4.0) + 1.0
  bind b = floor(beat % 4.0) + 1.0
  bind tick = round((beat % 1.0) * 960.0)
  return num_str(bar) + ":" + num_str(b) + ":" + num_str(tick)
}
fn snap_step(bw) {
  if bw >= 88.0 { return 0.25 }
  if bw >= 44.0 { return 0.5 }
  if bw >= 22.0 { return 1.0 }
  if bw >= 10.0 { return 2.0 }
  return 4.0
}
fn snap_to(v, s) { return round(v / s) * s }
bind start = do {
  bind pts = [[0.0, 120.0], [8.0, 90.0], [16.0, 140.0]]
  bind beats = [0.0, 4.0, 8.0, 12.0, 20.0]
  bind i = 0
  while i < len(beats) {
    bind b = list_get(beats, i)
```

```
    println("beat ", b, " = bar:beat:tick ", bar_beat(b), " = ", round(tm_secs(pts, b)
* 1000.0) / 1000.0, " s")
    bind i = i + 1
  }
  println("snap at 22 px/beat: ", snap_step(22.0), "  at 100 px/beat: ",
snap_step(100.0))
  println("drop at beat 3.37 snaps to ", snap_to(3.37, snap_step(22.0)), " (cell) and
", snap_to(3.37, 0.25), " (16th)")
}
```

```
OUTPUT
beat 0 = bar:beat:tick 1:1:0 = 0 s
beat 4 = bar:beat:tick 2:1:0 = 2 s
beat 8 = bar:beat:tick 3:1:0 = 4 s
beat 12 = bar:beat:tick 4:1:0 = 6.667 s
beat 20 = bar:beat:tick 6:1:0 = 11.048 s
snap at 22 px/beat: 1  at 100 px/beat: 0.25
drop at beat 3.37 snaps to 3 (cell) and 3.25 (16th)
```

How it works. A tempo point `[beat, bpm]` holds until the next one. For each segment that lies before the target beat, add `(b1 - b0) * 60 / bpm` seconds. The engine does the same: `daw_tempo_add(beat, bpm)` adds a point and `daw_secs_at(beat)` and `daw_bpm_at(beat)` read the map (the project recipe below calls them). `snap_step` is the function from `arranger.ling` with the zoom factor folded into pixels per beat: grid lines get denser as you zoom in, so a drop never snaps to a line you cannot see.

Variations. A 4/4 bar is four beats. For other meters, store the bar length per tempo point and replace the `4.0` constants. Ticks use 960 per beat, a common MIDI resolution.

ADSR envelope

An envelope shapes a note's loudness: attack to full level, decay to the sustain level, hold while the key is down, release after it is lifted. LingStationX's patches store the four values; its DSP scripts call the built-in `adsr(gate, attack, decay, sustain, release)`. This is a plain-Ling version, useful for drawing envelope editors and for offline rendering.

```
# ADSR as LingDSP's adsr(gate, a, d, s, r) shapes it: a piecewise-linear envelope sampled at time t
# note on at t=0, note off at t = held
fn adsr_at(t, held, a, d, s, r) {
  if t < held {
    if t < a { return t / a }
    if t < a + d { return 1.0 - (1.0 - s) * (t - a) / d }
    return s
  }
  # release from whatever level we were at when the key came up
  bind lvl = s
  if held < a { bind lvl = held / a }
  else if held < a + d { bind lvl = 1.0 - (1.0 - s) * (held - a) / d }
  bind u = (t - held) / r
  if u >= 1.0 { return 0.0 }
  return lvl * (1.0 - u)
}
fn bar(v) {
  bind n = round(v * 40.0)
  bind s = ""
  bind i = 0
  while i < n { bind s = s + "#"
    bind i = i + 1 }
  return s
}
bind start = do {
  bind t = 0.0
  while t <= 1.0001 {
    bind v = adsr_at(t, 0.6, 0.1, 0.2, 0.5, 0.3)
    println(format("{:.2} s  {:.2}  ", t, v) + bar(v))
    bind t = t + 0.1
  }
}
```

```
OUTPUT
0.00 s  0.00
0.10 s  1.00  ########################################
0.20 s  0.75  ##############################
0.30 s  0.50  ####################
0.40 s  0.50  ####################
0.50 s  0.50  ####################
0.60 s  0.50  ####################
0.70 s  0.33  #############
0.80 s  0.17  #######
0.90 s  0.00
1.00 s  0.00
```

How it works. While the key is down (`t < held`) the level follows attack and decay. After release the envelope starts from wherever it was, which matters when a key is lifted before the attack has finished, and falls linearly to zero over `r` seconds. The bars are 40 characters wide at full scale.

Variations. Replace the linear segments with `pow(x, 3)` for an exponential feel. The same function gives a filter envelope: multiply its output by the cutoff range instead of by amplitude.

Step sequencer and Euclidean rhythms

A step sequencer is a list of booleans. The Euclidean algorithm spreads *k* hits as evenly as possible over *n* steps and produces many traditional rhythms. This is the core of `scripts/euclid.ling`, which writes the result into a new one-bar clip.

```
# Euclidean rhythm, the test used by LingStationX's scripts/euclid.ling:
# step i is a hit when (i * pulses) % steps < pulses
fn euclid(pulses, steps) {
  bind out = []
  bind i = 0
  while i < steps {
    bind out = list_push(out, (i * pulses) % steps < pulses)
    bind i = i + 1
  }
  return out
}
fn row(name, hits, glyph) {
  bind s = name
  bind i = 0
  while i < len(hits) {
    if i % 4 == 0 { bind s = s + " " }
    if list_get(hits, i) { bind s = s + glyph } else { bind s = s + "." }
    bind i = i + 1
  }
  return s
}
bind start = do {
  println(row("kick  ", euclid(4, 16), "x"))
  println(row("snare ", euclid(2, 16), "o"))
  println(row("hat   ", euclid(7, 16), "-"))
  println(row("five  ", euclid(5, 16), "*"))
  # turn hits into [start, length, key, velocity] note records, a 16th = 0.25 beat
  bind notes = []
  bind hits = euclid(5, 16)
  bind i = 0
  while i < 16 {
    if list_get(hits, i) { bind notes = list_push(notes, [i * 0.25, 0.2, 36, 0.85]) }
    bind i = i + 1
  }
  println(notes)
}
```

```
OUTPUT
kick   x... x... x... x...
snare  o... .... o... ....
hat    -..- .-.- ..-. -.-.
five   *... *..* ..*. .*..
[[0, 0.2, 36, 0.85], [1, 0.2, 36, 0.85], [1.75, 0.2, 36, 0.85], [2.5, 0.2, 36, 0.85],
[3.25, 0.2, 36, 0.85]]
```

How it works. Step `i` is a hit when `(i * pulses) % steps < pulses`. Four hits over sixteen steps is four-on-the-floor; the other counts give the off-beat and syncopated figures found in many traditions. A hit becomes a note record `[start, length, key, velocity]`, with a sixteenth equal to 0.25 beats. In the real script the loop calls `daw_note_add(t, clip, start, length, key, velocity)` instead.

Variations. Add a rotation parameter by shifting the index: `((i + r) * pulses) % steps`. Layer several patterns on different drum keys (36 kick, 38 snare, 42 closed hat) in one clip.

Instrument patches as text

Because a patch is a text file, a parser is thirty lines. This recipe reads a Ling Synth X patch into a list of `[key, value]` pairs, edits one, and writes it back. The writer is the shape of LingStationX's "save patch" function.

```
# Instrument patches are plain text: "name = ..." header lines plus space separated
key=value pairs.
bind PATCH = "# Ling Synth X patch - edit freely
name = \"Init Saw\"
type = synth
wave=0.660 sub=1.000 add=0.000 fm=0.000 cutoff=0.600 reso=0.200 filt_env=0.400
partials=0.500 rolloff=0.500 odd/even=0.500 fm_ratio=0.250 fm_index=0.300 detune=0.150
noise=0.000 attack=0.050 decay=0.289 sustain=0.700 release=0.250 lfo_rate=0.300
lfo_cut=0.000"

fn patch_parse(text) {
  bind params = []
  bind meta = []
  bind lines = str_split(text, "\n")
  bind i = 0
  while i < len(lines) {
    bind ln = trim(list_get(lines, i))
    if str_len(ln) > 0 && starts_with(ln, "#") == false {
      if starts_with(ln, "name") || starts_with(ln, "type") {
        bind kv = str_split(ln, "=")
        bind meta = list_push(meta, [trim(list_get(kv, 0)),
trim(str_replace(list_get(kv, 1), "\"", ""))])
      } else {
        bind words = str_split(ln, " ")
        bind j = 0
        while j < len(words) {
          bind w = list_get(words, j)
          if str_find(w, "=") > 0 {
            bind kv = str_split(w, "=")
            bind params = list_push(params, [list_get(kv, 0), to_number(list_get(kv,
1))])
          }
          bind j = j + 1
        }
      }
    }
    bind i = i + 1
  }
```

```
  return [meta, params]
}
fn patch_get(params, key, dflt) {
  bind i = 0
  while i < len(params) {
    if list_get(list_get(params, i), 0) == key { return list_get(list_get(params, i),
1) }
    bind i = i + 1
  }
  return dflt
}
fn patch_write(meta, params) {
  bind out = "# Ling Synth X patch — edit freely\n"
  bind i = 0
  while i < len(meta) {
    bind m = list_get(meta, i)
    if list_get(m, 0) == "name" { bind out = out + "name = \"" + list_get(m, 1) +
"\"\n" }
    else { bind out = out + list_get(m, 0) + " = " + list_get(m, 1) + "\n" }
    bind i = i + 1
  }
  bind i = 0
  while i < len(params) {
    bind p = list_get(params, i)
    bind out = out + list_get(p, 0) + "=" + format("{:.3}", list_get(p, 1))
    if i % 6 == 5 { bind out = out + "\n" } else { bind out = out + " " }
    bind i = i + 1
  }
  return out
}
bind start = do {
  bind r = patch_parse(PATCH)
  bind meta = list_get(r, 0)
  bind params = list_get(r, 1)
  println("meta   ", meta)
  println("params ", len(params))
  println("cutoff ", patch_get(params, "cutoff", 0.5), "  release ", patch_get(params,
"release", 0.5), "  missing ", patch_get(params, "nope", -1))
  println("--- written back")
```

```
  bind text = patch_write(meta, params)
  println(text)
  bind again = patch_parse(text)
  println("round trip keeps ", len(list_get(again, 1)), " params")
}
```

```
OUTPUT
meta   [[name, Init Saw], [type, synth]]
params 20
cutoff 0.6  release 0.25  missing -1
--- written back
# Ling Synth X patch - edit freely
name = "Init Saw"
type = synth
wave=0.660 sub=1.000 add=0.000 fm=0.000 cutoff=0.600 reso=0.200
filt_env=0.400 partials=0.500 rolloff=0.500 odd/even=0.500 fm_ratio=0.250
fm_index=0.300
detune=0.150 noise=0.000 attack=0.050 decay=0.289 sustain=0.700 release=0.250
lfo_rate=0.300 lfo_cut=0.000
round trip keeps 20 params
```

How it works. Header lines (`name`, `type`) are metadata; every other non-comment line is split on spaces into `key=value` words. `str_find(w, "=") > 0` skips stray words. Dotted keys such as `kick.tune` survive because only the first `=` matters. The writer wraps six pairs per line so a patch stays diff-friendly.

Variations. Add a sanity check that every value lies in 0 to 1 and reject the file otherwise. For presets that morph, linearly interpolate two parsed patches key by key.

A synth voice rendered to a WAV file

Sometimes you want sound without the engine: a test tone, a sample for a game, a figure for a book. A voice is a function of time; a WAV file is a 44-byte header and 16-bit samples. Ling's byte helpers build both.

```
bind SR = 22050
fn midi_hz(k) { return 440.0 * pow(2.0, (k - 69.0) / 12.0) }
fn env(t, held, a, r) {
  if t < held {
    if t < a { return t / a }
    return 1.0
  }
  bind u = (t - held) / r
  if u >= 1.0 { return 0.0 }
  return 1.0 - u
}
# one voice: a saw built from 8 harmonics + a sub sine, shaped by an attack/release
envelope
fn voice(key, held, tail) {
  bind f = midi_hz(key)
  bind n = floor((held + tail) * SR)
  bind out = []
  bind i = 0
  while i < n {
    bind t = i / SR
    bind s = 0.0
    bind h = 1
    while h <= 8 {
      bind s = s + sin(2.0 * 3.14159265 * f * h * t) / h
      bind h = h + 1
    }
    bind s = (s * 0.5 + 0.3 * sin(3.14159265 * f * t)) * env(t, held, 0.01, tail)
    bind out = list_push(out, s)
    bind i = i + 1
  }
  return out
}
fn s16_hex(x) {
  bind v = round(x * 24000.0)
  if v > 32767 { bind v = 32767 }
  if v < -32768 { bind v = -32768 }
  if v < 0 { bind v = v + 65536 }
  return hex_slice(u32le_hex(v), 0, 2)
```

```
}
fn wav_hex(samples) {
  bind data = []
  bind i = 0
  while i < len(samples) {
    bind data = list_push(data, s16_hex(list_get(samples, i)))
    bind i = i + 1
  }
  bind body = hex_concat(data)
  bind n = hex_len(body)
  return hex_concat([text_hex("RIFF"), u32le_hex(36 + n), text_hex("WAVE"),
text_hex("fmt "), u32le_hex(16),
    "0100", "0100", u32le_hex(SR), u32le_hex(SR * 2), "0200", "1000", text_hex("data"),
u32le_hex(n), body])
}
bind start = do {
  bind s = voice(57, 0.3, 0.15)
  file_write_hex("voice.wav", wav_hex(s))
  println("samples: ", len(s), "  file bytes: ", hex_len(match
file_read_hex("voice.wav") { ok(v) => v, bad(e) => "" }))
  bind peak = 0.0
  bind i = 0
  while i < len(s) {
    if abs(list_get(s, i)) > peak { bind peak = abs(list_get(s, i)) }
    bind i = i + 1
  }
  println("peak amplitude: ", round(peak * 1000.0) / 1000.0)
  remove_file("voice.wav")
}
```

```
OUTPUT
samples: 9922  file bytes: 19888
peak amplitude: 0.89
```

How it works. `voice` sums eight sine harmonics scaled by 1/h (an approximation of a sawtooth), adds a sub-oscillator one octave down, and multiplies by an attack and release envelope. `s16_hex` clips to the 16-bit range and writes two little-endian bytes: `u32le_hex` of

the unsigned value, first two bytes kept. `wav_hex` concatenates the RIFF header, the `fmt` chunk (PCM, mono, 22050 Hz) and the data chunk, and `file_write_hex` writes it atomically.

Variations. Raise the harmonic count for a brighter sound, or replace the `sin` stack with a lookup table to speed up long renders. For stereo set the channel count to 2 and the byte rate and block align accordingly.

An instrument written in LingDSP, rendered and measured

LingStationX scripts can define instruments in LingDSP, a small expression language compiled by the engine (`scripts/supersaw.ling`, `fmbell.ling`, `karplus.ling`). Because `daw_bounce` renders offline, the whole loop (define, play, measure) runs without an audio device. This recipe builds a sine instrument, bounces two beats, then checks the pitch by counting zero crossings in the rendered file.

```
fn sine_dsp() {
    return "param level 0.5 0 1
env_ = adsr(gate, 0.01, 0.1, 0.8, 0.1)
out = osc(freq) * env_ * vel * level"
}
fn wav_left(hex, frame) {
    bind n = hex_u32le(hex, frame * 4).x
    bind s = n % 65536
    if s >= 32768 { return s - 65536 }
    return s
}
bind start = do {
  daw_new()
  daw_set_bpm(120.0)
  println("check: [", daw_script_check(sine_dsp(), "instrument"), "]")
  bind t = daw_script_inst_add("Sine", sine_dsp(), "")
  bind c = daw_clip_add_midi(t, 0.0, 2.0)
  daw_note_add(t, c, 0.0, 1.0, 69.0, 1.0)
  daw_bounce("sine.wav", 0.0, 2.0, 0.25, 16.0, "wav", 0.5)
  bind hex = match file_read_hex("sine.wav") { ok(v) => v, bad(e) => "" }
  println("frames ", (hex_len(hex) - 44) / 4)
  # analyse 0.2 s from t = 0.1 s (frames 4800..14400 at 48 kHz); slice first, hex ops
are linear in length
  bind win = hex_slice(hex, 44 + 4800 * 4, 9600 * 4)
  bind peak = 0
  bind cross = 0
  bind prev = 0
  bind i = 0
  while i < 9600 {
    bind s = wav_left(win, i)
    if abs(s) > peak { bind peak = abs(s) }
    if prev < 0 && s >= 0 { bind cross = cross + 1 }
    bind prev = s
    bind i = i + 1
  }
  println("peak ", peak, " rising zero crossings in 0.2 s: ", cross)
  println("estimated Hz: ", cross * 5, "  (the note was A4 = 440 Hz; resolution is 5
```

```
Hz)")
  remove_file("sine.wav")
}
```

```
OUTPUT
check: []
frames 60000
peak 5418 rising zero crossings in 0.2 s: 87
estimated Hz: 435  (the note was A4 = 440 Hz; resolution is 5 Hz)
```

How it works. A LingDSP program sees `freq`, `vel`, `gate` and `t` for the note being played. `param name default min max` lines become knobs. `daw_script_check` returns an empty string when the program compiles, otherwise the error text. `daw_script_inst_add(label, source, path)` creates the track; `daw_bounce(path, start, end, tail, bits, format, quality)` writes 48 kHz stereo audio. The analysis slices a 0.2 second window out of the file's hex first because hex operations are linear in the string length.

Variations. Count zero crossings over a longer window for a finer estimate, or take the maximum absolute sample to assert that a patch never clips. The same bounce can be a regression test: render, hash the bytes with `lw_h256`, compare to a stored hash.

Project API, script-style note edits, undo and MIDI export

This is the full lifecycle of a clip using the `daw_*` builtins, with a transformation written the way LingStationX scripts write them: read the flat note list, build a new one, set it. The recipe checks that undo and redo restore the original, and exports the track as a Standard MIDI File.

```
# the shape of every Ling Station script: <name>_run(track, clip) edits notes through daw_* calls
fn nudge(notes, amount) {
  bind out = []
  bind n = len(notes) / 7
  bind i = 0
  while i < n {
    bind b = i * 7
    # deterministic "humanize": alternate pushing late and early, and soften every other note
    bind off = amount
    if i % 2 == 1 { bind off = 0.0 - amount }
    bind out = list_push(out, max(0.0, list_get(notes, b) + off))
    bind out = list_push(out, list_get(notes, b + 1))
    bind out = list_push(out, list_get(notes, b + 2))
    bind v = list_get(notes, b + 3)
    if i % 2 == 1 { bind v = v * 0.8 }
    bind out = list_push(out, v)
    bind out = list_push(out, list_get(notes, b + 4))
    bind out = list_push(out, list_get(notes, b + 5))
    bind out = list_push(out, list_get(notes, b + 6))
    bind i = i + 1
  }
  return out
}
fn show(label, notes) {
  bind n = len(notes) / 7
  bind s = label + ": "
  bind i = 0
  while i < n {
    bind s = s + format("[{:.2}@{:.0} v{:.2}] ", list_get(notes, i * 7), list_get(notes, i * 7 + 2), list_get(notes, i * 7 + 3))
    bind i = i + 1
  }
  println(s)
}
bind start = do {
  daw_new()
```

```
  daw_set_bpm(100.0)
  bind t = daw_track_add("synth", "")
  bind c = daw_clip_add_midi(t, 0.0, 4.0)
  bind i = 0
  while i < 4 {
    daw_note_add(t, c, i * 1.0, 0.9, 60 + i * 2, 0.9)
    bind i = i + 1
  }
  show("original ", daw_clip_notes(t, c))
  daw_undo_push()
  daw_clip_set_notes(t, c, nudge(daw_clip_notes(t, c), 0.05))
  show("humanized", daw_clip_notes(t, c))
  println("can undo: ", daw_can_undo())
  daw_undo()
  show("undone   ", daw_clip_notes(t, c))
  println("can redo: ", daw_can_redo())
  daw_redo()
  show("redone   ", daw_clip_notes(t, c))
  daw_tempo_add(8.0, 60.0)
  println("bpm at beat 4: ", daw_bpm_at(4.0), "  at beat 9: ", daw_bpm_at(9.0))
  println("seconds at beat 12: ", round(daw_secs_at(12.0) * 1000.0) / 1000.0)
  println("export: ", daw_midi_export(t, "idea.mid"), "  ", file_size("idea.mid"), "
bytes")
  remove_file("idea.mid")
}
```

```
OUTPUT
original : [0.00@60 v0.90] [1.00@62 v0.90] [2.00@64 v0.90] [3.00@66 v0.90]
humanized: [0.05@60 v0.90] [0.95@62 v0.72] [2.05@64 v0.90] [2.95@66 v0.72]
can undo: 1
undone   : [0.00@60 v0.90] [1.00@62 v0.90] [2.00@64 v0.90] [3.00@66 v0.90]
can redo: 1
redone   : [0.05@60 v0.90] [0.95@62 v0.72] [2.05@64 v0.90] [2.95@66 v0.72]
bpm at beat 4: 100  at beat 9: 60
seconds at beat 12: 8.8
export: 1  80 bytes
```

How it works. `daw_undo_push()` marks the state before an edit; `daw_undo()` returns to it and `daw_redo()` goes forward again. Predicates such as `daw_can_undo()` return numbers (0 or 1), not booleans, so print them or compare with `> 0.5`. `daw_clip_notes` returns seven numbers per note, so note `i` starts at index `i * 7`. The humanizer here is deterministic (alternating early and late) so that its output is reproducible; the shipped `humanize.ling` uses `rand()`.

Variations. Save the project with `daw_project_save(path)` and reopen it with `daw_project_load(path)`. Wrap the edit in an `x_info()` / `x_run(track, clip)` pair of functions named after the file to turn it into a LingStationX script.

Mixer maths: decibels, meters and pan

A channel strip is three numbers: gain, pan, and a meter. The meter in `core.ling` maps a linear peak to a bar between -60 and 0 dB. Constant-power pan keeps loudness steady as a sound moves across the stereo field.

```
fn db_to_gain(db) { return pow(10.0, db / 20.0) }
fn gain_to_db(g) { if g < 0.0001 { return -80.0 }
  return 20.0 * log10(g) }
# meter fill 0..1 from a linear peak, -60 dB .. 0 dB (LingStationX's m_level)
fn meter_fill(a) {
  if a < 0.001 { return 0.0 }
  bind f = (20.0 * log10(a) + 60.0) / 60.0
  if f < 0.0 { return 0.0 }
  if f > 1.0 { return 1.0 }
  return f
}
# constant-power pan: p in -1..1 -> [left, right] gains
fn pan_gains(p) {
  bind ang = (p + 1.0) * 0.25 * 3.14159265
  return [cos(ang), sin(ang)]
}
# meter ballistics: instant attack, exponential release (call once per frame)
fn meter_step(shown, peak, dt) {
  if peak > shown { return peak }
  return shown * pow(0.5, dt / 0.15)
}
bind start = do {
  println("-6 dB = gain ", round(db_to_gain(-6.0) * 1000.0) / 1000.0, "; gain 0.5 = ",
round(gain_to_db(0.5) * 100.0) / 100.0, " dB")
  bind a = [1.0, 0.5, 0.1, 0.01, 0.001]
  bind i = 0
  while i < len(a) {
    bind n = round(meter_fill(list_get(a, i)) * 30.0)
    bind bar = ""
    bind j = 0
    while j < n { bind bar = bar + "="
      bind j = j + 1 }
    println(format("peak {:.3} -> fill {:.2} |", list_get(a, i), meter_fill(list_get(a,
i))) + bar)
    bind i = i + 1
  }
  bind p = -1.0
  while p <= 1.01 {
```

```
    bind g = pan_gains(p)
    println(format("pan {:.1}: L {:.3}  R {:.3}  power {:.3}", p, list_get(g, 0),
list_get(g, 1), list_get(g, 0) * list_get(g, 0) + list_get(g, 1) * list_get(g, 1)))
    bind p = p + 0.5
  }
  bind shown = 1.0
  bind k = 0
  while k < 5 {
    bind shown = meter_step(shown, 0.0, 0.1)
    bind k = k + 1
  }
  println("after 0.5 s of silence the needle is at ", round(shown * 1000.0) / 1000.0)
}
```

```
OUTPUT
-6 dB = gain 0.501; gain 0.5 = -6.02 dB
peak 1.000 -> fill 1.00 |==============================
peak 0.500 -> fill 0.90 |==========================
peak 0.100 -> fill 0.67 |====================
peak 0.010 -> fill 0.33 |==========
peak 0.001 -> fill 0.00 |
pan -1.0: L 1.000  R 0.000  power 1.000
pan -0.5: L 0.924  R 0.383  power 1.000
pan 0.0: L 0.707  R 0.707  power 1.000
pan 0.5: L 0.383  R 0.924  power 1.000
pan 1.0: L 0.000  R 1.000  power 1.000
after 0.5 s of silence the needle is at 0.099
```

How it works. `20 * log10(g)` converts linear gain to decibels. The meter clamps `(dB + 60) / 60` to 0 to 1. The pan law `L = cos(a)`, `R = sin(a)` with `a` from 0 to a quarter turn makes `L² + R² = 1`, so centre is -3 dB per side rather than a 6 dB bump. `meter_step` gives the needle instant attack and an exponential release with a 0.15 s half-life; call it once per frame with the frame time.

Variations. For a peak-hold line, keep a second value that only decays after one second. For a loudness meter, average squared samples over 300 ms before taking the logarithm.

Piano roll geometry and hit testing

A piano roll is a grid whose horizontal axis is beats and whose vertical axis is MIDI keys, with key 127 at the top. All the mouse behaviour (drag, resize, select) reduces to converting between pixels and `(beat, key)` and asking which note owns that cell.

```
# piano roll geometry: beats -> x, MIDI key -> y (key 127 at the top), and the hit test
bind PX_BEAT = 6.0
bind PX_KEY = 2.0
fn beat_x(b) { return b * PX_BEAT }
fn key_y(k, top) { return (top - k) * PX_KEY }
fn hit_note(notes, bx, ky) {
  bind i = 0
  while i < len(notes) {
    bind n = list_get(notes, i)
    if bx >= list_get(n, 0) && bx < list_get(n, 0) + list_get(n, 1) && floor(ky) ==
list_get(n, 2) { return i }
    bind i = i + 1
  }
  return -1
}
bind start = do {
  bind notes = [[0.0, 2.0, 60], [2.0, 2.0, 64], [4.0, 2.0, 67], [6.0, 1.0, 72], [7.0,
1.0, 71]]
  bind lo = 60
  bind hi = 72
  bind k = hi
  while k >= lo {
    bind line = ""
    if k % 12 == 1 || k % 12 == 3 || k % 12 == 6 || k % 12 == 8 || k % 12 == 10 { bind
line = "#  " } else { bind line = "   " }
    bind line = format("{:.0}", k) + " " + line
    bind c = 0
    while c < 8 {
      bind ch = "."
      bind i = 0
      while i < len(notes) {
        bind n = list_get(notes, i)
        if list_get(n, 2) == k && c >= list_get(n, 0) && c < list_get(n, 0) +
list_get(n, 1) { bind ch = "=" }
        bind i = i + 1
      }
      bind line = line + ch + ch
      bind c = c + 1
```

```
    }
    println(line)
    bind k = k - 1
  }
  println("click at x=" + num_str(beat_x(3.0)) + " px (beat 3), key 64: note ",
hit_note(notes, 3.0, 64.5))
  println("click on empty cell: ", hit_note(notes, 1.0, 70.5))
  println("key 64 sits at y = ", key_y(64, 84), " px below the top of an 84-key
window")
}
```

```
OUTPUT
72    ...........==..
71    ............==
70 #  ..............
69    ..............
68 #  ..............
67    ........====....
66 #  ..............
65    ..............
64    ....====........
63 #  ..............
62    ..............
61 #  ..............
60    ====............
click at x=18 px (beat 3), key 64: note 1
click on empty cell: -1
key 64 sits at y = 40 px below the top of an 84-key window
```

How it works. `beat_x(b) = b * px_per_beat` and `key_y(k, top) = (top - k) * px_per_key`. The inverse maps a mouse position to `floor(key)` and a fractional beat. `hit_note` returns the index of the first note whose rectangle contains the point, or -1. The ASCII picture shows black-key rows marked with `#`; the real roll darkens those lanes.

Variations. For resizing, test whether the click is within 4 pixels of a note's right edge. For large clips cache the note list by `daw_rev()` as `arranger.ling` does and draw them with one `px_notes` call.

Drawing a piano roll with the pixel builtins

The same geometry drawn for real. `px_rect`, `px_hline`, `px_vline`, `px_rrect` and `px_text` take packed colours (`r * 65536 + g * 256 + b`), and `px_rrect` takes an optional last argument for alpha, which this recipe uses to turn velocity into brightness. `px_flat()` is called once at start and the font is an ordinary TrueType file.

GUI

```
# colours are one packed number: r * 65536 + g * 256 + b
fn rgb(r, g, b) { return r * 65536 + g * 256 + b }
bind BG = rgb(26, 28, 40)
bind LANE_BLACK = rgb(16, 18, 28)
bind GRID = rgb(44, 48, 64)
bind NOTE = rgb(80, 200, 120)

fn black_key(k) {
  bind p = k % 12
  return p == 1 || p == 3 || p == 6 || p == 8 || p == 10
}

fn draw_roll(font, x, y, w, h, notes, lo, hi, bw) {
  bind kh = h / (hi - lo + 1)
  px_rect(x, y, w, h, BG)
  # key lanes: black-key rows are darker, C rows get a label
  bind k = lo
  while k <= hi {
    bind ky = y + (hi - k) * kh
    if black_key(k) { px_rect(x, ky, w, kh, LANE_BLACK) }
    px_hline(x, ky, w, GRID)
    if k % 12 == 0 { px_text(font, x + 4, ky + 1, kh - 2, 9474192, "C" + num_str(k / 12
- 1)) }
    bind k = k + 1
  }
  # beat lines, bar lines every 4 beats
  bind b = 0
  while b * bw < w {
    px_vline(x + b * bw, y, h, sel_col(b % 4 == 0))
    bind b = b + 1
  }
  # notes: start, length, key, velocity per entry
  bind i = 0
  while i < len(notes) {
    bind n = list_get(notes, i)
    bind nx = x + list_get(n, 0) * bw
    bind ny = y + (hi - list_get(n, 2)) * kh
```

```
    px_rrect(nx + 1, ny + 1, list_get(n, 1) * bw - 2, kh - 2, 2, NOTE, 0.4 + 0.6 *
list_get(n, 3))
    bind i = i + 1
  }
}
fn sel_col(major) { if major { return 8421504 }
  return 4210752 }

bind start = do {
  open_window(640, 360, "Piano roll")
  px_flat()
  bind font = font_load("font.ttf")
  bind notes = [[0, 2, 60, 0.9], [2, 2, 64, 0.7], [4, 2, 67, 0.8], [6, 1, 72, 1.0], [7,
1, 71, 0.6], [0, 8, 48, 0.5]]
  bind frame = 0
  while window_is_open() && frame < 13 {
    fill(10, 12, 20)
    draw_roll(font, 20, 20, 600, 320, notes, 46, 74, 72)
    present()
    bind frame = frame + 1
    if frame == 10 { screenshot("roll") }
  }
}
```



Figure 10.3. The output of the recipe above: black-key lanes, beat and bar lines, and six notes with velocity as opacity.

How it works. The function draws background, lane stripes, grid, then notes, in that order, because there is no z-buffer; later calls cover earlier ones. Everything is positioned from one `bw` (pixels per beat) and a derived key height, so zooming is just a different `bw` and `hi - lo`. The `while` loop with `frame < 13` replaces a `break`, which Ling does not have, and `screenshot("roll")` writes a PNG into `screenshots/`.

Variations. Add `px_clip(x, y, w, h)` and `px_unclip()` around the call to keep the roll inside a panel. Put the lanes in a loop over `st_get("rsel")` to draw selection outlines, and read `mouse_x()` and `mouse_down()` inside the same function to make it interactive.

CHAPTER 14

Crypto Recipes: A Privacy Wallet in Ling

Ling Wallet is a post-quantum privacy wallet, a full node and a miner for an experimental coin, LNG, in one application written entirely in Ling: about 8,800 lines, no foreign-function glue. This chapter explains how it is structured and then gives nine recipes for the cryptographic chores every such program has: hashing, byte layouts, address encoding, encrypted storage, signatures, key derivation, key exchange, isolating secrets and proof of work.

> **WARNING**
>
> LNG is a new, unaudited coin and the wallet's ring signature and proof of work are novel. The recipes below demonstrate sound habits with standard primitives. They are not a substitute for an audit, and nothing here should be used to protect funds you cannot afford to lose.

Snippets marked `ling` were run with `ling run --interp` and the output shown is real. They use only builtins that exist in the Ling runtime: the `lw_*` family in `src/runtime/lwcrypto.rs`, the byte helpers in `src/runtime/bytes.rs`, and a few general builtins (`sha256_hex`, `shake_hex`, `file_write_hex`). No network connection was opened and no miner was started while the snippets were run.

How a privacy wallet is structured in Ling

Modules

Folder	Lines	Contents
`ling/main.ling`	181	Entry point and frame loop; `--mock` runs the UI against simulated data
`ling/backend.ling`, `real_backend.ling`, `mock.ling`	101, 535, 274	The `be_*` contract between UI and engine (`be_create`, `be_unlock`, `be_prepare`, `be_confirm`, ...)

Folder	Lines	Contents
`ling/core/`	2,230	Consensus and wallet logic with no I/O: `codec` (byte layouts), `money`, `params`, `chain` (blocks, validation), `mempool`, `wallet` (keys, scanning, transactions)
`ling/isolates/`	1,480	Three long-running threads: `node`, `walletd`, `miner`
`ling/ui/`	3,460	One module per screen plus `theme`, `widgets`, `input`, `qr`, `torus`
`ling/tests/`		Byte-compatibility tests for the crypto builtins, chain, node, p2p and thread tests; `selftest.ling` drives the real backend with no window

Separating UI, node, wallet and miner

The most important design decision is that the program is four isolates, not one. In Ling an *isolate* is an interpreter on its own OS thread. Nothing is shared except channels and a write-once store, and each isolate is spawned with an explicit **capability list**:

```
OUTPUT
node      caps: db, fs, p2p     chain database, mempool, network
walletd   caps: keys, db, fs    the only place secret keys exist
miner     caps: (none)          can only hash and talk over its channels
UI        main thread           draws, never sees a key
```

A builtin whose capability was not granted fails with `capability "keys" not granted`, and a child can never receive a capability its parent lacks. Secret keys never enter Ling as values at all: `lw_spend_keypair` and friends return an opaque, random 128-bit **handle**, and the key material stays in native memory behind it, zeroed when `lw_key_drop` is called. Even a bug in the UI or a malicious script run in the main thread cannot print a key it was never given. "Keeping secrets in one isolate" shows the pattern.

The UI talks to the engine only through the `be_*` functions (`docs/LING_BACKEND.md`), all of them non-blocking: the UI sends a request on a channel, and a later `be_tick()` call drains replies. The same contract is implemented twice, by `real_backend.ling` and by `mock.ling`, which is how the screenshots in this book were made without a node.

Storage

- **The wallet file** (`wallet.lw`) holds one thing: the 32-byte master seed, plus a birth height and a list of notes. It is encrypted with XChaCha20-Poly1305 under a key derived from the password with Argon2id (256 MiB, three passes by default). The header is bound into the ciphertext as authenticated data. "Encrypted storage" below is a simplified version.

- **The chain database** (`chain.db`) is SQLite through the `db_*` builtins, in write-ahead-log mode, and is public data: blocks and transactions. There are no plaintext balances or history on disk; the wallet recomputes them by scanning.
- **Settings** are a few lines of text. Nothing secret lives in them.

UI screens

The desktop UI is a state machine over screens: onboarding (create or restore, password, 24-word recovery phrase with a re-entry quiz), unlock, **Wallet** (balance and activity), **Send** (compose, review, authorize, broadcast), **Receive** (address, fingerprint, QR), **Mining** (the background miner with a live 4-D torus visualisation), **Network** (peers, supply audit) and **Security** (recovery phrase, auto-lock, password change). Sending forces the user to compare a recipient fingerprint and re-enter the password before each payment.

Domain-separated hashing

Never hash raw data with a bare function. If a transaction hash and a memo hash can collide on the same bytes, an attacker will find a way to present one as the other. `lw_h256(domain, parts)` hashes a *domain string* together with a list of hex parts, so the same input hashed for two purposes yields unrelated outputs. `lw_xof` squeezes an arbitrary number of bytes.

```
bind start = do {
  bind msg = text_hex("pay 5 LNG to alice")
  bind a = lw_h256("demo/tx", [msg])
  bind b = lw_h256("demo/memo", [msg])
  println("tx   ", a)
  println("memo ", b)
  println("same input, different domain: ", hex_eq(a, b))
  println("deterministic: ", hex_eq(a, lw_h256("demo/tx", [msg])))
  println("xof 8 bytes: ", lw_xof("demo/derive", [msg], 8))
  # general-purpose hashes of text, for checksums and interop (not domain separated)
  println("sha256 of abc: ", sha256_hex("abc"))
  println("shake256 of abc, 16 bytes: ", shake_hex("abc", 16))
}
```

```
OUTPUT
tx   152cb7f1865b1c7930d6545f347a423a25a3de359454cc09dd1507742beeafaa
memo 9993441b4bb50f7579ec710af62f18f18cfbccb1124362e5f9575783315da4e6
same input, different domain: false
deterministic: true
xof 8 bytes: 41c717518aff6c8a
sha256 of abc: ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
shake256 of abc, 16 bytes: 483366601360a8771c6863080cc4114d
```

How it works. Bytes always cross the Ling boundary as lowercase hex strings, so `text_hex("...")` turns text into the form the builtins take. `lw_h256` is SHA-3 based; the wallet uses a distinct domain for every use (`LW1/bundle`, `LW1/address`, `LW1/derive/spend`). `sha256_hex` and `shake_hex` are available for interoperability, for example to publish a file checksum, but they are not domain separated.

Variations. Hash several fields with one call by passing a list: `lw_h256("tx/v1", [a, b, c])`. The parts are not simply concatenated: `["ab", "c"]` and `["a", "bc"]` hash differently, which plain concatenation would not guarantee (checked: `hex_eq` of the two hashes is `false`).

Security. Choose domain strings once, include a version number, and never reuse one for another purpose. Compare hashes with `hex_eq`, which is constant time, not `==`.

Binary records with the byte helpers

Consensus data needs one exact byte layout. The wallet builds every record by concatenating fixed-width fields as hex and reads them back by offset. All the helpers are strict: odd-length hex, non-hex characters, and values at or above 2^53 are errors.

```
bind start = do {
  bind rec = hex_concat([u8_hex(1), u32le_hex(300), u64le_hex(5000000000),
varint_hex(300), text_hex("hi")])
  println(rec)
  println("len ", hex_len(rec))
  println("version ", hex_u8(rec, 0))
  bind n = hex_u32le(rec, 1)
  println(n)
  match hex_u64le(rec, 5) {
    ok(v) => println("amount ", v),
    bad(e) => println("bad ", e),
  }
  match hex_varint(rec, 13) {
    ok(p) => println("varint ", list_get(p, 0), " next ", list_get(p, 1)),
    bad(e) => println("bad ", e),
  }
  match hex_text(hex_slice(rec, 15, 2)) {
    ok(t) => println("text ", t),
    bad(e) => println("bad ", e),
  }
  println(hex_u32le("0102", 0))
}
```

```
OUTPUT
012c01000000f2052a01000000ac026869
len 17
version Ok(1)
Ok(300)
amount 5000000000
varint 300 next 15
text hi
Err(read past end of data)
```

How it works. `u8_hex`, `u32le_hex`, `u64le_hex` and `varint_hex` write little-endian and LEB128 fields; `hex_u8`, `hex_u32le`, `hex_u64le` and `hex_varint` read them at a byte offset and return `ok(value)` (printed `Ok(1)`) or `bad(reason)` (printed `Err(...)`). `hex_varint` returns `[value,`

`next_offset]`. The last line shows a read past the end: it fails cleanly instead of reading garbage. The wallet's parser wraps these in `rd_begin`, `rd_u64` and `rd_end` helpers with a sticky failure flag, so a truncated file is detected once at the end.

Variations. Use `hex_slice(h, start, len)` for fixed-size fields such as keys and `hex_concat([...])` to join. For amounts above 2^53 use the `big_add`, `big_sub` and `big_cmp` helpers, which take decimal strings.

Security. Parse untrusted input only through these helpers, check lengths before slicing, and cap every count you read from the data (the wallet caps note counts and string lengths) before looping over it.

Address encoding with a checksum

People type addresses, and people make typos. The wallet's short address is `lng1` followed by the base32 encoding of a 32-byte hash and a 4-byte checksum. A mistyped character is rejected before any money moves. This is `short_address` and `parse_short_address` from `core/wallet.ling`, with a demo domain string.

```
fn checksum(h) { return hex_slice(lw_h256("demo/addr-checksum", [text_hex("lng1"), h]),
0, 4) }
fn encode_addr(h) { return "lng1" + base32_encode(hex_concat([h, checksum(h)])) }
fn decode_addr(s) {
  bind t = trim(s)
  if starts_with(t, "lng1") == false { return bad("addresses start with lng1") }
  bind raw = match base32_decode(substr(t, 4)) { ok(v) => v, bad(e) => "" }
  if hex_len(raw) != 36 { return bad("wrong length or invalid characters") }
  bind h = hex_slice(raw, 0, 32)
  if hex_eq(hex_slice(raw, 32, 4), checksum(h)) == false { return bad("checksum
mismatch (typo?)") }
  return ok(h)
}
bind start = do {
  bind h = lw_h256("demo/pubkey", [text_hex("alice")])
  bind a = encode_addr(h)
  println(a)
  println(decode_addr(a))
  bind typo = substr(a, 0, 10) + "q" + substr(a, 11)
  println(typo)
  println(decode_addr(typo))
  println(decode_addr("btc1xyz"))
}
```

OUTPUT

```
lng13ikags34agruoe6nlvi2rvw7u2oju4kmdym5llomprmmjdpeje6qfjs6ry
Ok(da14034b7c01a34713cd5d51a8d6dfa69c9a714c1e19d5adcc7c58c48de4493d)
lng13ikagsq4agruoe6nlvi2rvw7u2oju4kmdym5llomprmmjdpeje6qfjs6ry
Err(checksum mismatch (typo?))
Err(addresses start with lng1)
```

How it works. The address is a hash of the recipient's full public key bundle, not the keys themselves, so it stays short (62 characters here: the prefix, then 36 bytes in base32). The checksum is the first four bytes of a domain-separated hash. `base32_encode` produces RFC 4648 lowercase without padding; `base32_decode` rejects anything non-canonical. Decoding returns `ok(hash)` or `bad(reason)`, never a half-valid value. The wallet then fetches the full signed bundle from the network and accepts it only if its hash matches.

Variations. The full bundle is exported as `lngkey1` plus `base64url_encode(bundle_hex)`. Use `hex_len` to validate the decoded length (36 bytes here) before slicing.

Security. A checksum protects against accidents, not attackers: an attacker can generate a valid address. The wallet therefore shows a fingerprint, `xxxx-xxxx-xxxx-xxxx`, derived from the key, and asks the user to compare it with the recipient over a second channel.

Encrypted storage: Argon2id plus an AEAD

A wallet file must be useless without the password and must reveal tampering. The recipe derives a key with Argon2id (`lw_password_key`), encrypts with XChaCha20-Poly1305 (`lw_aead_seal`), and stores the KDF parameters, salt and nonce in an authenticated header. It then proves three properties: it round-trips, a wrong password fails, and one flipped bit fails.

```
bind MAGIC = "4c575731"
fn seal_file(path, password, secret_hex) {
  bind salt = lw_random(16)
  bind nonce = lw_random(24)
  bind m_kib = 4096
  bind t_cost = 1
  bind key = match lw_password_key(password, salt, m_kib, t_cost, 1) { ok(k) => k,
bad(e) => "" }
  if key == "" { return bad("kdf failed") }
  bind header = hex_concat([MAGIC, u32le_hex(m_kib), u32le_hex(t_cost), salt, nonce])
  bind ct = lw_aead_seal(key, nonce, header, secret_hex)
  wipe(key)
  return file_write_hex(path, hex_concat([header, ct]))
}
fn open_file(path, password) {
  bind data = match file_read_hex(path) { ok(v) => v, bad(e) => "" }
  if hex_len(data) < 48 || hex_slice(data, 0, 4) != MAGIC { return bad("not a vault
file") }
  bind m_kib = hex_u32le(data, 4).x
  bind t_cost = hex_u32le(data, 8).x
  if m_kib > 1048576 || t_cost > 16 { return bad("refusing absurd KDF parameters") }
  bind key = match lw_password_key(password, hex_slice(data, 12, 16), m_kib, t_cost, 1)
{ ok(k) => k, bad(e) => "" }
  bind header = hex_slice(data, 0, 52)
  bind opened = lw_aead_open(key, hex_slice(data, 28, 24), header, hex_slice(data, 52,
hex_len(data) - 52))
  wipe(key)
  return match opened { ok(p) => ok(p), bad(e) => bad("wrong password or damaged file")
}
}
bind start = do {
  bind secret = lw_random(32)
  println("saved: ", seal_file("vault.lw", "correct horse battery staple", secret))
  println("size on disk: ", hex_len(match file_read_hex("vault.lw") { ok(v) => v,
bad(e) => "" }))
  bind back = open_file("vault.lw", "correct horse battery staple")
  bind plain = match back { ok(p) => p, bad(e) => "" }
  println("round trip ok: ", hex_eq(plain, secret))
```

```
  println("wrong password: ", open_file("vault.lw", "hunter2"))
  bind raw = match file_read_hex("vault.lw") { ok(v) => v, bad(e) => "" }
  bind flipped = hex_concat([hex_slice(raw, 0, 60), u8_hex(255 - hex_u8(raw, 60).x),
hex_slice(raw, 61, hex_len(raw) - 61)])
  file_write_hex("tampered.lw", flipped)
  println("tampered file: ", open_file("tampered.lw", "correct horse battery staple"))
  remove_file("vault.lw")
  remove_file("tampered.lw")
}
```

```
OUTPUT
saved: Ok(())
size on disk: 100
round trip ok: true
wrong password: Err(wrong password or damaged file)
tampered file: Err(wrong password or damaged file)
```

How it works. The file is `magic | m_kib | t_cost | salt(16) | nonce(24) | ciphertext+tag`. The whole 52-byte header is passed to `lw_aead_seal` as *associated data*: it is not encrypted but it is authenticated, so changing the Argon2 parameters or the salt invalidates the tag. `lw_aead_open` returns `ok(plaintext)` or `bad`; the recipe deliberately gives the same message for a wrong password and for damage, so the file does not become a password oracle. The demo uses 4 MiB of memory so it runs instantly; Ling Wallet uses 256 MiB.

Variations. Prepend a version byte to the magic so the format can evolve. Write with `file_write_hex`, which is atomic (a temporary file, then a rename), so a crash never leaves half a wallet. Store the KDF cost in the header and re-encrypt on unlock when a better setting is configured.

Security.

- Draw a fresh salt and nonce from `lw_random` on every save. Reusing a nonce with the same key breaks XChaCha20-Poly1305.
- **Check the header before using it.** `wallet_load` refuses absurd parameters (`m_kib > 4 GiB`, `t_cost > 64`) so a crafted file cannot exhaust memory.
- Refuse to overwrite a wallet with one that has no seed. `wallet_save` returns `bad` unless the master is exactly 32 bytes, because a save after the keys were dropped by a lock once would otherwise erase the only copy.

- Call `wipe(secret)` after use. Be aware of its limit: the runtime documentation states that `wipe` is currently best effort, because builtins receive copies of their arguments. The reliable defence is not to hold secrets in Ling strings for long, which is what handles are for.

Signatures: identity keys with ML-DSA-87

An identity key signs things other people must trust, such as the wallet's address bundle. `lw_id_keypair` takes a 32-byte seed and returns a handle; `lw_id_sign` signs a message under a *context* string; `lw_id_verify` needs no secret and returns a plain boolean. Verifiers answer `false` to any malformed input rather than raising.

```
bind start = do {
  bind h = match lw_id_keypair(lw_random(32)) { ok(k) => k, bad(e) => "" }
  bind pk = lw_id_public(h)
  println("public key bytes: ", hex_len(pk))
  println("fingerprint is 16 hex digits in 4 groups: ", str_len(lw_id_fingerprint(pk))
== 19)
  bind msg = lw_h256("demo/doc", [text_hex("transfer 5 LNG")])
  bind sig = lw_id_sign(h, "demo-sign", msg)
  println("signature bytes: ", hex_len(sig))
  println("verifies: ", lw_id_verify(pk, "demo-sign", msg, sig))
  println("wrong context: ", lw_id_verify(pk, "other", msg, sig))
  bind other = lw_h256("demo/doc", [text_hex("transfer 500 LNG")])
  println("altered message: ", lw_id_verify(pk, "demo-sign", other, sig))
  println("garbage: ", lw_id_verify("00", "demo-sign", msg, sig))
  println("dropped: ", lw_key_drop(h))
  println("dropped twice: ", lw_key_drop(h))
}
```

```
OUTPUT
public key bytes: 2592
fingerprint is 16 hex digits in 4 groups: true
signature bytes: 4627
verifies: true
wrong context: false
altered message: false
garbage: false
dropped: true
dropped twice: false
```

How it works. ML-DSA-87 is the NIST post-quantum signature scheme in its strongest parameter set: 2592-byte public keys and 4627-byte signatures. The context is part of the signed data (`H("LW1/sig/" + context, msg)`), so a signature made for `"bundle"` cannot be replayed as a signature for `"transaction"`: the "wrong context" line fails. Altering one byte of the message fails too. `lw_key_drop` zeroizes the key and returns `true` the first time, `false` afterwards.

Variations. Sign the *hash* of a large message, as here, not the message itself. Publish `lw_id_fingerprint(pk)` instead of the full key for humans to compare. Ring signatures for payments are `lw_ring_sign` and `lw_ring_verify`; they are experimental and described in the wallet's design document, not here.

Security. Always verify before parsing the signed content, never after. A signature proves who signed, not that the content is true.

From a recovery phrase to independent subkeys

One 32-byte master seed backs up the whole wallet. It is shown to the user as 24 words (BIP-39 English), and every key is derived from it with `lw_xof` under a distinct domain. Compromising one derived key reveals nothing about the others.

```
bind start = do {
  # a fixed demo seed; a real wallet draws it from lw_random(32) and never prints it
  bind master = "000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f"
  bind words = lw_mnemonic_from(master)
  println(len(words), " words; first three: ", list_get(words, 0), " ", list_get(words,
1), " ", list_get(words, 2))
  bind phrase = join(words, " ")
  bind back = lw_mnemonic_to(phrase)
  bind got = match back { ok(v) => v, bad(e) => "" }
  println("restores: ", hex_eq(got, master))
  println("bad word count: ", lw_mnemonic_to("abandon abandon abandon"))
  bind s1 = lw_xof("demo/derive/spend", [master], 32)
  bind s2 = lw_xof("demo/derive/view", [master], 32)
  println("independent subkeys: ", hex_eq(s1, s2) == false)
}
```

```
OUTPUT
24 words; first three: abandon amount liar
restores: true
bad word count: Err(a recovery phrase has exactly 24 words)
independent subkeys: true
```

How it works. `lw_mnemonic_from(hex32)` returns a list of 24 words; `lw_mnemonic_to(text)` lowercases, collapses whitespace, checks the word count and the checksum, and returns `ok(hex32)` or `bad(reason)`. The seed in the recipe is the standard test vector `000102...1f`, whose phrase starts "abandon amount liar". The wallet derives its keys in `keys_load`:

```
OUTPUT
spend_seed = lw_xof("LW1/derive/spend",    [master], 32)
view_seed  = lw_xof("LW1/derive/view",     [master], 96)
id_seed    = lw_xof("LW1/derive/identity", [master], 32)
```

Each seed goes into the matching `lw_*_keypair` call, and the handle is kept while the seed string is wiped.

Variations. Add a per-account index to the domain or the parts list for multiple accounts. Verify the phrase the user wrote down by asking for three random words, as the wallet's re-entry quiz does.

Security. The 24 words are the wallet. Show them once, on a screen that hides itself after a timeout, require the password to show them again, and never log or screenshot them. Never use a fixed seed like the demo one for anything real.

Hybrid key exchange

To send a message or a payment note that only the recipient can read, the sender needs a shared secret. The wallet uses a hybrid of X25519 and ML-KEM-1024: an attacker must break both, so recorded traffic stays safe even after quantum computers arrive. `lw_kem_encaps` runs on the sender's side with only the public key; `lw_kem_decaps` runs on the recipient's with the secret handle.

```
bind start = do {
  bind rk = match lw_kem_keypair(lw_random(96)) { ok(k) => k, bad(e) => "" }
  bind pk = lw_kem_public(rk)
  println("kem public key bytes: ", hex_len(pk))
  bind enc = match lw_kem_encaps(pk, text_hex("demo-ctx")) { ok(r) => r, bad(e) => [] }
  bind ct = list_get(enc, 0)
  bind ss_sender = list_get(enc, 1)
  println("ciphertext bytes: ", hex_len(ct))
  bind ss_recv = match lw_kem_decaps(rk, ct, text_hex("demo-ctx")) { ok(s) => s, bad(e)
=> "" }
  println("both sides agree: ", hex_eq(ss_sender, ss_recv))
  bind wrong = match lw_kem_decaps(rk, ct, text_hex("other-ctx")) { ok(s) => s, bad(e)
=> "" }
  println("other context agrees: ", hex_eq(ss_sender, wrong))
  lw_key_drop(rk)
}
```

```
OUTPUT
kem public key bytes: 1600
ciphertext bytes: 1600
both sides agree: true
other context agrees: false
```

How it works. The recipient generates a keypair from 96 random bytes and publishes `lw_kem_public`. The sender calls `lw_kem_encaps(pk, context)`, which returns a ciphertext to transmit and a 32-byte shared secret to keep. The recipient recovers the same secret from the ciphertext. The context is mixed into the derivation, so decapsulating with a different context yields a different secret (the last line). The shared secret then keys `lw_aead_seal`.

Variations. The wallet derives one-time output keys from the shared secret and an output index (`lw_onetime_public`), which is how every payment lands on a fresh address.

Security. Treat the shared secret like a key: derive purpose-specific keys from it with `lw_xof` rather than using it directly, and never reuse a nonce with it.

Keeping secrets in one isolate

The strongest key-handling rule is structural: secrets live in exactly one place, and everyone else asks that place to act. Here a signer isolate is spawned with the `keys` capability. It owns the identity handle; the main thread holds only the public key and sends it a hash to sign. The signature is verified outside the signer.

```
fn signer_main(req, rep) {
  bind h = match lw_id_keypair(lw_random(32)) { ok(k) => k, bad(e) => "" }
  chan_send(rep, ["pk", lw_id_public(h)])
  while true {
    match chan_recv(req) {
      ok(m) => {
        if list_get(m, 0) == "stop" { lw_key_drop(h)
          return ok(()) }
        chan_send(rep, ["sig", lw_id_sign(h, "demo", list_get(m, 1))])
      },
      bad(e) => { return ok(()) },
    }
  }
  return ok(())
}
bind start = do {
  bind req = chan_new(4)
  bind rep = chan_new(4)
  bind t = thread_spawn("cr_isolate.ling", "signer_main", [list_get(req, 1),
list_get(rep, 0)], ["keys"])
  bind pk = match chan_recv(list_get(rep, 1)) { ok(m) => list_get(m, 1), bad(e) => "" }
  bind msg = lw_h256("demo/m", [text_hex("hello")])
  chan_send(list_get(req, 0), ["sign", msg])
  bind sig = match chan_recv(list_get(rep, 1)) { ok(m) => list_get(m, 1), bad(e) => ""
}
  println("verified outside the signer: ", lw_id_verify(pk, "demo", msg, sig))
  chan_send(list_get(req, 0), ["stop"])
  println(thread_join(match t { ok(h) => h, bad(e) => "" }))
}
```

```
OUTPUT
verified outside the signer: true
Ok(Ok(()))
```

How it works. `thread_spawn(file, function, args, capabilities)` starts the function in its own interpreter. `chan_new(capacity)` returns `[send_handle, recv_handle]`; values sent over a channel are deep-copied, and functions cannot be sent. The signer receives a message, signs,

replies, and drops its key on `"stop"`. `thread_join` returns the function's result wrapped in `ok`. Key handles are valid only inside the isolate that made them.

Variations. `chan_new_secure(n)` zeroizes string payloads after delivery, for channels that carry passwords or seeds. Give a hashing worker `[]` capabilities: the miner cannot read a file, open a socket or create a key.

Security. This is the least-privilege principle applied to a language runtime. Give every worker the smallest capability set it needs, and review any change that widens one.

Proof of work and a capability check

Mining is hashing a header with a changing nonce until the result is below a target. LNG uses a memory-hard algorithm, HyperTorus, so `lw_pow_hash` keeps a 16 MiB scratchpad per isolate. The recipe checks two things the wallet relies on: pure compute needs no capability, but creating a key does.

```
fn probe() {
  return lw_id_keypair(lw_random(32))
}
bind start = do {
  bind t = match thread_spawn("cr_pow.ling", "probe", [], []) { ok(h) => h, bad(e) =>
"" }
  println(thread_join(t))
  bind t2 = match thread_spawn("cr_pow.ling", "probe", [], ["keys"]) { ok(h) => h,
bad(e) => "" }
  println(match thread_join(t2) { ok(v) => "got a key handle", bad(e) => e })
  bind header = hex_concat([text_hex("demo-block"), u64le_hex(1)])
  bind pow = lw_pow_hash(hex_concat([header, u64le_hex(42)]))
  println("pow hash bytes: ", hex_len(pow))
  println("meets difficulty 1: ", lw_meets_difficulty(pow, "1"))
  println("meets huge difficulty: ", lw_meets_difficulty(pow,
"340282366920938463463374607431768211455"))
}
```

```
OUTPUT
Err(capability "keys" not granted)
got a key handle
pow hash bytes: 32
meets difficulty 1: true
meets huge difficulty: false
```

How it works. The first spawned isolate has no capabilities and `lw_id_keypair` is refused with `capability "keys" not granted`; the second is granted `keys` and succeeds. `lw_pow_hash(header_hex)` returns a 32-byte hash. `lw_meets_difficulty(hash_hex, difficulty)` takes a decimal difficulty up to 128 bits (a larger value is a runtime error). The real miner builds `header ++ u64le_hex(nonce)`, hashes, tests, increments, and sleeps between hashes to honour an intensity setting; and the UI can pause it while the computer is in use or on battery (`sys_idle_secs`, `sys_on_battery`).

Variations. `lw_pow_trace(header_hex, every)` also returns sampled coordinates of the four-dimensional walk, which the Mining screen draws.

Security guidance for the whole chapter.

1. Keep secrets in handles, in one capability-limited isolate; never in globals or the shared store.
2. Use a fresh random salt and nonce from `lw_random` every time; never from `rand()`, which is not a cryptographic source.
3. Authenticate everything you store or transmit (an AEAD with the header as associated data), and verify before parsing.
4. Bound every size and count that comes from outside before allocating or looping.
5. Compare secrets and hashes with `hex_eq`.
6. Re-ask for the password before irreversible actions, and auto-lock after idle time.
7. Say plainly what is unaudited. The wallet's `THREAT_MODEL.md` lists what is standard, what is novel and what is experimental, and the UI repeats it on the About screen.

CHAPTER 15

Tooling Recipes: Editor Apps

An editor application is a small operating system: panels that resize, a selection, tools, an undo stack, files that must survive a crash, and a viewport that redraws sixty times a second. LingDev (a scene, timeline and cutscene editor for games, about 28,000 lines of Ling) and LingModeler (a 3D modeler, vector illustration, animation and procedural-texture studio, about 16,000 lines) are both written entirely in Ling and share one design. This chapter describes that design and gives eleven recipes adapted from the two programs.

Snippets marked `ling` were run with `ling run --interp` and the output shown is real. The snippet marked `ling-gui` opens a window and was run once to produce its screenshot.

Architecture of an editor in Ling

Layout of the source

Both editors follow the module pattern of the LingStationX DAW from the previous chapter (`core.ling` input and widgets, `ds.ling` design system, `shell.ling` menus, `actions.ling` string-keyed command dispatch), then add modules for their own domain.

LingDev module	Role
`app.ling`	`draw_frame()`: lays out hierarchy, assets, viewport, inspector and dock
`project.ling`	`lingdev.toml`, scenes on disk, undo, selection, project templates
`rt/` (19 modules, 18,500 lines)	The **runtime**: scene model, key evaluation, easing, cutscenes, 3D software renderer, game modes. Shipped inside every exported game
`timeline.ling`, `cutscene.ling`	Keyframe layers and six cutscene tracks
`inspector.ling`, `hier.ling`, `assets.ling`	Property editor, scene tree, asset browser
`view3d.ling`, `view2d.ling`	Viewports with gizmos
`buildpanel.ling`, `importers.ling`	`ling build` front end, importers for existing games

LingModeler module	Role
`mesh.ling`, `meshops.ling`	Scene objects, polygon meshes, selection, undo; extrude, bevel, loop cut, subdivide
`rig.ling`, `pose.ling`, `anim.ling`	Bones, weights, skinning, keyframed actions
`vec2d.ling`, `svg.ling`	Bezier paths, layers, SVG import and export
`texture.ling`, `raster.ling`	16 procedural generators and a software rasteriser
`lingfmt.ling`, `objfmt.ling`, `gltf.ling`	Native `.lmesh`, OBJ and glTF I/O

The decisive choice in LingDev is that **the editor and the game share one runtime**. The scene model, the parser and the serializer live in `rt/rt_scene.ling` and are `use`d by both the editor and the exported game, so "what you see in the editor" and "what the player sees" cannot drift apart. The editor even plays the game inside its own window (F5) by calling the runtime's `game_update` and `game_render`.

Panels and tool state

The frame function computes every panel's rectangle from the window size and a handful of saved numbers (`left_w`, `insp_w`, `dock_h`), clamps them so no panel can swallow the viewport, then calls each panel with `(x, y, w, h)`. A splitter is a six-pixel hot zone: on press it sets the store key `active` to its name, and while the key is active and the button is down, it writes the mouse position into the matching width. Tool state (the current tool, grid snap, auto-key) lives in `st_*` keys like all other state, and `do_action("edit.undo")`-style strings are the only way menus, toolbars and shortcuts reach the code.

Files: two formats you can read in a text editor

- `lingdev.toml` is a flat `key=value` file; a repeated key (`scene=...`) lists the project's scenes. A project folder also holds `scenes/*.lscene`, `scripts/*.ling`, `assets/` and a copy of `rt/`.
- `.lscene` is one record per line, fields separated by `|`: `scene|name=...`, `node|id=4|name=Hero|px=0|...`, `key|n=5|p=px|t=6|v=-3|e=sine_in_out`, `clip|tr=camera|t=0|d=6|k=orbit|...`. Because it is line-based, version control shows readable diffs and one corrupt line loses one node, not the scene.
- `.lmesh` is more surprising: it is **valid Ling source**. Each datum is a one-line function (`fn lm_o0_name() { return "Tree" }`, `fn lm_o0_vx() { return lm_nums("0,1,0.5") }`), plus generated `draw_Tree(ox, oy, oz, scale)` functions. A game can `use "hero.lmesh"` and call the draw function with no loader at all, and the editor reads the file back line by line.

Undo, and why both editors snapshot

Neither editor records inverse operations. LingDev serializes the whole scene to a string and pushes it on a list (see "Undo and redo with snapshots"). LingModeler copies every scene key under a prefix (`u0_`, `u1_`, ...) in the store. Because Ling values are immutable, the copies share their lists until one side is edited, which is why a snapshot of a large mesh is cheap. Snapshots make every operation undoable for free, including the ones added later.

Test hooks

Both programs accept `--shot N` (screenshot at frame N, then exit), `--do "frame,action;..."` and `--drive "frame,x,y,down;..."` (scripted mouse). LingModeler's `tools/make_screenshots.py` uses them to regenerate every screenshot in its README, and LingStationX's `tools/smoke.py` opens every dialog, tab and menu the same way.

Undo and redo with snapshots

LingDev keeps two lists in the native store, `undo` and `redo`, of whole-scene strings. An edit pushes the state *before* the edit and clears the redo list; undo pushes the present onto redo and restores the last undo entry.

```
# Undo / redo as a stack of serialized snapshots (the LingDev design).
# The document is one string; every edit pushes the string as it was BEFORE the edit.
fn doc_text() { return st_get("doc", "") }
fn undo_push() {
  st_lpush("undo", doc_text())
  while st_llen("undo") > 3 { st_ldel("undo", 0) }
  st_lclear("redo")
}
fn undo_do() {
  bind n = st_llen("undo")
  if n < 1 { return "Nothing to undo" }
  st_lpush("redo", doc_text())
  st_set("doc", st_lget("undo", n - 1))
  st_ldel("undo", n - 1)
  return "Undo"
}
fn redo_do() {
  bind n = st_llen("redo")
  if n < 1 { return "Nothing to redo" }
  st_lpush("undo", doc_text())
  st_set("doc", st_lget("redo", n - 1))
  st_ldel("redo", n - 1)
  return "Redo"
}
fn edit(text) {
  undo_push()
  st_set("doc", text)
}
fn show(msg) { println(format("{} doc=\"{}\"  undo={} redo={}", msg, doc_text(),
st_llen("undo"), st_llen("redo"))) }
bind start = do {
  st_set("doc", "")
  show("start")
  edit("a")
  show("edit a")
  edit("ab")
  show("edit ab")
  edit("abc")
```

```
  show("edit abc")
  show(undo_do())
  show(undo_do())
  show(redo_do())
  edit("abX")
  show("edit abX")
  show(redo_do())
  edit("1")
  edit("12")
  edit("123")
  edit("1234")
  show("four more edits")
  show(undo_do())
  show(undo_do())
  show(undo_do())
  show(undo_do())
}
```

```
OUTPUT
start doc=""  undo=0 redo=0
edit a doc="a"  undo=1 redo=0
edit ab doc="ab"  undo=2 redo=0
edit abc doc="abc"  undo=3 redo=0
Undo doc="ab"  undo=2 redo=1
Undo doc="a"  undo=1 redo=2
Redo doc="ab"  undo=2 redo=1
edit abX doc="abX"  undo=3 redo=0
Nothing to redo doc="abX"  undo=3 redo=0
four more edits doc="1234"  undo=3 redo=0
Undo doc="123"  undo=2 redo=1
Undo doc="12"  undo=1 redo=2
Undo doc="1"  undo=0 redo=3
Nothing to undo doc="1"  undo=0 redo=3
```

How it works. `st_lpush(list, v)`, `st_llen(list)`, `st_lget(list, i)`, `st_ldel(list, i)` and `st_lclear(list)` are the store's list helpers. The real stack holds 60 entries and drops the oldest. The run shows the usual surprises: after undoing and then making a new edit, redo is cleared; after four rapid edits only the last three remain.

Variations.

- **Coalesce a drag.** In the inspector, a draggable number calls `undo_push()` once, on the press that starts the drag (`if hot && u_pressed() { undo_push() }`), not on every frame of it. A text field pushes once per focus, remembered in an `insp_undo_fid` key, so typing a word is one undo step.
- **Mark dirty.** `undo_push` also sets a `dirty` flag the title bar and the autosave timer read (see "Autosave").
- **Keep selection out of the snapshot,** then prune it after a restore (`sel_prune`) so it cannot point at a node that no longer exists.

A line-based record format

Define the file format first; the rest of an editor is easier after that. A record is `kind|key=value|key=value`; unknown keys are ignored and missing keys take defaults, so old files open in new versions and the reverse mostly works. This is the core of `sc_parse` and `sc_serialize`.

```
# Line records "kind|key=value|key=value" (the .lscene scene format) and a key=value
project file.
fn rec_get(fields, key, dflt) {
  bind head = key + "="
  bind i = 1
  while i < len(fields) {
    bind f = list_get(fields, i)
    if starts_with(f, head) { return substr(f, str_len(head)) }
    bind i = i + 1
  }
  return dflt
}
fn rec_num(fields, key, dflt) {
  bind v = rec_get(fields, key, "")
  if v == "" { return dflt }
  return to_number(v)
}
fn fld(key, v) { return "|" + key + "=" + str_replace(str_replace(v, "|", "/"), "\n", "
") }
fn fldn(key, v) { return "|" + key + "=" + num_str(v) }

fn scene_parse(text) {
  bind nodes = []
  bind keys = []
  bind lines = str_split(text, "\n")
  bind i = 0
  while i < len(lines) {
    bind ln = trim(list_get(lines, i))
    if ln != "" {
      bind f = str_split(ln, "|")
      bind kind = list_get(f, 0)
      if kind == "node" {
        bind nodes = list_push(nodes, [rec_num(f, "id", 0), rec_get(f, "name", ""),
rec_num(f, "px", 0), rec_num(f, "py", 0)])
      } else if kind == "key" {
        bind keys = list_push(keys, [rec_num(f, "n", 0), rec_get(f, "p", ""),
rec_num(f, "t", 0), rec_num(f, "v", 0), rec_get(f, "e", "linear")])
      }
```

```
    }
    bind i = i + 1
  }
  return [nodes, keys]
}
fn scene_write(nodes, keys) {
  bind s = "lingdev-scene|v=1\n"
  bind i = 0
  while i < len(nodes) {
    bind n = list_get(nodes, i)
    bind s = s + "node" + fldn("id", list_get(n, 0)) + fld("name", list_get(n, 1)) +
fldn("px", list_get(n, 2)) + fldn("py", list_get(n, 3)) + "\n"
    bind i = i + 1
  }
  bind i = 0
  while i < len(keys) {
    bind k = list_get(keys, i)
    bind s = s + "key" + fldn("n", list_get(k, 0)) + fld("p", list_get(k, 1)) +
fldn("t", list_get(k, 2)) + fldn("v", list_get(k, 3)) + fld("e", list_get(k, 4)) + "\n"
    bind i = i + 1
  }
  return s
}
bind SCENE = "lingdev-scene|v=1
scene|name=Cutscene|kind=cutscene|fps=30|len=16
node|id=4|name=Hero|px=0|py=-1
node|id=5|name=Spirit|px=3|py=-2
key|n=5|p=px|t=0|v=3|e=linear
key|n=5|p=px|t=6|v=-3|e=sine_in_out
"
bind start = do {
  bind r = scene_parse(SCENE)
  println("nodes: ", list_get(r, 0))
  println("keys:  ", list_get(r, 1))
  bind text = scene_write(list_get(r, 0), list_get(r, 1))
  print_file(text)
  bind again = scene_parse(text)
  println("round trip stable: ", scene_write(list_get(again, 0), list_get(again, 1)) ==
text)
```

```
  # a field value can never smuggle in a record separator
  println(fld("name", "evil|node|id=9"))
}
```

```
OUTPUT
nodes: [[4, Hero, 0, -1], [5, Spirit, 3, -2]]
keys:  [[5, px, 0, 3, linear], [5, px, 6, -3, sine_in_out]]
lingdev-scene|v=1
node|id=4|name=Hero|px=0|py=-1
node|id=5|name=Spirit|px=3|py=-2
key|n=5|p=px|t=0|v=3|e=linear
key|n=5|p=px|t=6|v=-3|e=sine_in_out
round trip stable: true
|name=evil/node/id=9
```

How it works. `rec_get` and `rec_num` scan a record's fields for `key=`. `fld` writes a text value after replacing `|` and newlines, so no value can smuggle in a record separator (the last line of output); `fldn` writes a number. The round-trip check serializes the parsed data and compares it with the text. The `.lscene` loader in LingDev is this loop with a column list (`nd_cols()`) instead of four hard-coded fields.

Variations. Put a version in the first line (`lingdev-scene|v=1`) and branch on it. Write the file with a temporary name and a rename (see "Autosave"). For the project file use the same idea without `|`: `cfg_get(text, "fps", "60")` is a loop over lines of `key=value`.

Keyframes with per-key easing

Both editors animate by storing sparse keys `[time, value, easing]` and evaluating them at the playhead. The easing belongs to the key at the *end* of a segment: it says how to arrive. This is `key_eval` from `rt_scene.ling` and a few curves from `rt_ease.ling`, which LingDev writes in Ling so that they run identically on desktop, web and the LingOS kernel.

```
# Keyframes with per-key easing (rt_ease.ling + key_eval from LingDev's runtime)
fn ease(kind, t) {
  if t < 0.0 { return 0.0 }
  if t > 1.0 { return 1.0 }
  if kind == "step" { if t >= 1.0 { return 1.0 }
    return 0.0 }
  if kind == "quad_in" { return t * t }
  if kind == "cubic_out" { return 1.0 - pow(1.0 - t, 3.0) }
  if kind == "sine_in_out" { return 0.0 - (cos(3.14159265 * t) - 1.0) / 2.0 }
  if kind == "cubic_in_out" {
    if t < 0.5 { return 4.0 * t * t * t }
    return 1.0 - pow(0.0 - 2.0 * t + 2.0, 3.0) / 2.0
  }
  return t
}
# keys: [time, value, easing-into-this-key], sorted by time
fn key_eval(keys, t, dflt) {
  bind n = len(keys)
  if n < 1 { return dflt }
  if t <= list_get(list_get(keys, 0), 0) { return list_get(list_get(keys, 0), 1) }
  bind last = list_get(keys, n - 1)
  if t >= list_get(last, 0) { return list_get(last, 1) }
  bind i = 0
  while i < n - 1 {
    bind a = list_get(keys, i)
    bind b = list_get(keys, i + 1)
    if t >= list_get(a, 0) && t <= list_get(b, 0) {
      bind u = (t - list_get(a, 0)) / (list_get(b, 0) - list_get(a, 0))
      return list_get(a, 1) + (list_get(b, 1) - list_get(a, 1)) * ease(list_get(b, 2),
u)
    }
    bind i = i + 1
  }
  return list_get(last, 1)
}
bind start = do {
  bind keys = [[0.0, 3.0, "linear"], [6.0, -3.0, "sine_in_out"], [8.0, 0.0, "step"]]
  bind t = 0.0
```

```
  while t <= 8.01 {
    bind v = key_eval(keys, t, 0.0)
    bind col = round((v + 3.0) / 6.0 * 30.0)
    bind line = ""
    bind j = 0
    while j < col { bind line = line + " "
      bind j = j + 1 }
    println(format("t={:.1} x={:.2} ", t, v) + line + "*")
    bind t = t + 1.0
  }
}
```

```
OUTPUT
t=0.0 x=3.00                                *
t=1.0 x=2.60                               *
t=2.0 x=1.50                          *
t=3.0 x=0.00                  *
t=4.0 x=-1.50         *
t=5.0 x=-2.60   *
t=6.0 x=-3.00 *
t=7.0 x=-3.00 *
t=8.0 x=0.00                  *
```

How it works. Before the first key and after the last, the value holds. Inside a segment, `u = (t - ta) / (tb - ta)` is eased and used to interpolate. The sample run shows three behaviours: a linear first approach, a smooth sine-in-out slide from x = 3 to x = -3, and a `step` key, which stays at -3 until the instant of the last key and then jumps to 0.

Variations. Interpolate angles the short way round (`ease_angle` in the runtime wraps differences into -pi to pi). A property with no keys returns its stored value, so non-animated nodes cost nothing. For a 21-curve set add the `bounce`, `expo`, `back` and `elastic` families from the same file.

Timeline scrubber maths

A timeline needs three conversions (time to pixel, pixel to time, time to frame), a zoom that keeps the time under the mouse fixed, and a ruler whose labels never crowd each other. This is `tl_px`, `tl_time`, `tl_snap_time` and the zoom code from `timeline.ling`.

```
# Timeline math: seconds <-> pixels, zoom about the cursor, frame snapping, ruler ticks
fn t_px(t, gx, off, zoom) { return gx + (t - off) * zoom }
fn px_t(px, gx, off, zoom) { return off + (px - gx) / zoom }
fn snap_frame(t, fps) {
  bind s = round(t * fps) / fps
  if s < 0.0 { return 0.0 }
  return s
}
# zoom by wheel steps, keeping the time under the mouse fixed
fn zoom_at(mx, gx, off, zoom, wheel) {
  bind mt = px_t(mx, gx, off, zoom)
  bind nz = zoom * pow(1.15, wheel)
  if nz < 8.0 { bind nz = 8.0 }
  if nz > 2000.0 { bind nz = 2000.0 }
  bind noff = mt - (mx - gx) / nz
  if noff < 0.0 { bind noff = 0.0 }
  return [noff, nz]
}
# tick spacing in seconds that stays at least 50 px apart
fn tick_step(zoom) {
  bind steps = [0.05, 0.1, 0.25, 0.5, 1.0, 2.0, 5.0, 10.0, 30.0, 60.0]
  bind i = 0
  while i < len(steps) {
    if list_get(steps, i) * zoom >= 50.0 { return list_get(steps, i) }
    bind i = i + 1
  }
  return 60.0
}
fn mmss(t) {
  bind m = floor(t / 60.0)
  bind s = t - m * 60.0
  return format("{:.0}:{:.2}", m, s)
}
bind start = do {
  bind gx = 190.0
  bind off = 0.0
  bind zoom = 60.0
  println("t=2.5 s is at x=", t_px(2.5, gx, off, zoom), "; x=400 is t=", px_t(400.0,
```

```
gx, off, zoom), " s")
  println("scrub to 1.234 s at 30 fps snaps to ", round(snap_frame(1.234, 30.0) *
10000.0) / 10000.0, " s (frame ", round(snap_frame(1.234, 30.0) * 30.0), ")")
  bind z = zoom_at(400.0, gx, off, zoom, 6.0)
  println("zoom in 6 wheel clicks: zoom ", round(list_get(z, 1) * 10.0) / 10.0, " px/s,
offset ", round(list_get(z, 0) * 1000.0) / 1000.0, " s")
  println("time under mouse before ", px_t(400.0, gx, off, zoom), " after ",
px_t(400.0, gx, list_get(z, 0), list_get(z, 1)))
  println("tick every ", tick_step(60.0), " s at 60 px/s; every ", tick_step(10.0), " s
at 10 px/s; every ", tick_step(600.0), " s at 600 px/s")
  println("playhead label: ", mmss(75.5))
}
```

```
OUTPUT
t=2.5 s is at x=340; x=400 is t=3.5 s
scrub to 1.234 s at 30 fps snaps to 1.2333 s (frame 37)
zoom in 6 wheel clicks: zoom 138.8 px/s, offset 1.987 s
time under mouse before 3.5 after 3.5
tick every 1 s at 60 px/s; every 5 s at 10 px/s; every 0.1 s at 600 px/s
playhead label: 1:15.50
```

How it works. `px = gx + (t - off) * zoom`, where `off` is the time at the left edge and `zoom` is pixels per second. To zoom about the mouse, convert the mouse x to a time first, change the zoom, then solve for the new `off` so the same time sits under the same pixel (the two printed "time under mouse" values match). `tick_step` picks the smallest step from a fixed list that is at least 50 pixels wide.

Variations. Snap the playhead to frames with `round(t * fps) / fps`; the output shows frame 37 at 30 fps. Clamp `off` at zero so the user cannot scroll before the start.

Drawing and dragging a timeline strip

The same maths as drawing code: a ruler, key diamonds, a playhead you drag, wheel zoom. Every pixel comes from `px_rect`, `px_vline`, `px_text`, and `draw_poly` after `px_color`, which draws a filled polygon in the current colour.

GUI

```
# An interactive timeline strip: ruler, key diamonds, a draggable playhead, wheel zoom.
fn rgb(r, g, b) { return r * 65536 + g * 256 + b }
fn tick_step(zoom) {
  bind steps = [0.05, 0.1, 0.25, 0.5, 1.0, 2.0, 5.0, 10.0, 30.0]
  bind i = 0
  while i < len(steps) {
    if list_get(steps, i) * zoom >= 50.0 { return list_get(steps, i) }
    bind i = i + 1
  }
  return 60.0
}
fn diamond(cx, cy, r, c) {
  px_color(c)
  draw_poly([cx, cy - r, cx + r, cy, cx, cy + r, cx - r, cy])
}
fn timeline(font, x, y, w, keys, t, off, zoom) {
  px_rect(x, y, w, 22, rgb(36, 38, 52))
  px_rect(x, y + 22, w, 60, rgb(22, 24, 34))
  bind step = tick_step(zoom)
  bind s = floor(off / step) * step
  while (s - off) * zoom < w {
    bind tx = x + (s - off) * zoom
    px_vline(tx, y + 14, 8, rgb(150, 150, 170))
    px_text(font, tx + 3, y + 3, 11, rgb(170, 170, 190), num_str(s) + "s")
    bind s = s + step
  }
  # one diamond per key, on the layer row
  bind i = 0
  while i < len(keys) {
    diamond(x + (list_get(keys, i) - off) * zoom, y + 52, 6, rgb(240, 200, 60))
    bind i = i + 1
  }
  # playhead
  bind px = x + (t - off) * zoom
  px_vline(px, y, 82, rgb(240, 80, 80))
  px_rect(px - 4, y, 9, 6, rgb(240, 80, 80))
}
```

```
bind start = do {
  open_window(640, 200, "Timeline")
  px_flat()
  bind font = font_load("font.ttf")
  bind t = 2.0
  bind zoom = 60.0
  bind frame = 0
  while window_is_open() && frame < 13 {
    fill(12, 14, 22)
    # drag the playhead anywhere on the strip; the wheel zooms
    if mouse_down() && mouse_y() < 130 && mouse_y() > 40 { bind t = max(0.0, (mouse_x()
- 20) / zoom) }
    bind zoom = max(8.0, min(400.0, zoom * pow(1.15, mouse_scroll())))
    timeline(font, 20, 40, 600, [0.5, 2.0, 3.5, 6.0, 8.0], t, 0.0, zoom)
    px_text(font, 20, 140, 14, rgb(200, 200, 220), "t = " + format("{:.2}", t) + " s
zoom = " + format("{:.0}", zoom) + " px/s")
    present()
    bind frame = frame + 1
    if frame == 10 { screenshot("scrub") }
  }
}
```



Figure 12.1. The strip at the first frame: playhead at 2.0 s, keys at 0.5, 2, 3.5, 6 and 8 s.

How it works. Input and drawing share one function: the code that draws the strip also reads `mouse_x()` and `mouse_down()` for the drag, which is how LingDev's panels work (the hit test is the rectangle just drawn). `mouse_scroll()` returns the wheel delta for the frame. The loop ends after 13 frames and `screenshot("scrub")` saves a PNG, because Ling has no `break`.

Variations. Add `px_clip(x, y, w, h)` so labels cannot overflow the panel, and `px_hline` / `px_vline` rows for each layer.

Dock layout with resizable panels

LingDev's window is five rectangles computed from the window size and three user-set numbers, each clamped so the panels cannot fight. This is the core of `draw_frame()` in `app.ling`, returning rectangles instead of drawing.

```
# Dock layout the way LingDev's draw_frame does it: fixed bars, resizable side panels and dock,
# every size clamped so no panel can eat the viewport. Returns rectangles [x, y, w, h].
fn clampf(v, lo, hi) { if v < lo { return lo }
  if v > hi { return hi }
  return v }
fn layout(W, H, left_w, insp_w, dock_h) {
  bind top = 24.0 + 28.0 + 34.0
  bind bottom = H - 22.0
  bind lw = clampf(left_w, 180.0, W * 0.4)
  bind iw = clampf(insp_w, 220.0, W * 0.4)
  bind dh = clampf(dock_h, 120.0, (bottom - top) * 0.7)
  bind right = W - iw
  bind vbottom = bottom - dh
  return [
    ["hierarchy", 0.0, top, lw, bottom - top],
    ["inspector", right, top, iw, bottom - top],
    ["dock", lw, vbottom, right - lw, dh],
    ["viewport", lw, top, right - lw, vbottom - top]
  ]
}
fn show(rects) {
  bind i = 0
  while i < len(rects) {
    bind r = list_get(rects, i)
    println(format("{} x={:.0} y={:.0} w={:.0} h={:.0}", list_get(r, 0), list_get(r, 1), list_get(r, 2), list_get(r, 3), list_get(r, 4)))
    bind i = i + 1
  }
}
bind start = do {
  println("1360 x 820, defaults")
  show(layout(1360.0, 820.0, 260.0, 300.0, 240.0))
  println("same window, user drags every splitter too far")
  show(layout(1360.0, 820.0, 900.0, 50.0, 900.0))
}
```

```
OUTPUT
1360 x 820, defaults
hierarchy x=0 y=86 w=260 h=712
inspector x=1060 y=86 w=300 h=712
dock x=260 y=558 w=800 h=240
viewport x=260 y=86 w=800 h=472
same window, user drags every splitter too far
hierarchy x=0 y=86 w=544 h=712
inspector x=1140 y=86 w=220 h=712
dock x=544 y=300 w=596 h=498
viewport x=544 y=86 w=596 h=214
```

How it works. Bars have fixed heights (the numbers in the snippet are illustrative, not LingDev's); the hierarchy and inspector widths are clamped to between a minimum and 40 percent of the window; the dock height is clamped to 70 percent of the free area. The viewport gets what is left. The second run drags every splitter to an extreme and the layout stays valid. A splitter is a six-pixel hot zone straddling the edge: pressing it sets `u_set_active("left_resize")`, and while that key is active and the button is down, the new width is `u_mx()`.

Variations. Save the three numbers in `settings.txt` (`left_w`, `insp_w`, `dock_h`) so the layout persists. A `show_hier` flag per panel collapses it by changing its width to zero before clamping.

A table-driven property inspector

An inspector has dozens of fields, and writing one block of UI code per field does not scale. LingDev describes each as a row of data (column name, label, range, drag speed, format) and one function draws or edits any row. In the real inspector `prop_num` also draws a key diamond beside each number, turning the field into a keyframe.

```
# Property inspector driven by a table: one row spec per property, one loop draws or
edits them all.
# row = [column, label, min, max, drag speed, format]
fn rows() {
  return [["px", "X", -100.0, 100.0, 0.05, "num"], ["py", "Y", -100.0, 100.0, 0.05,
"num"],
          ["rx", "Rot X", -3.14159, 3.14159, 0.01, "deg"], ["sx", "Scale X", 0.01,
100.0, 0.02, "num"], ["vis", "Visible", 0.0, 1.0, 1.0, "int"]]
}
fn fmt_value(v, f) {
  if f == "deg" { return format("{:.1}", v * 57.29578) + " deg" }
  if f == "int" { return num_str(floor(v)) }
  return format("{:.2}", v)
}
fn clampf(v, lo, hi) { if v < lo { return lo }
  if v > hi { return hi }
  return v }
# dragging by dx pixels changes the value by dx * speed, clamped to the row's range
fn drag(v, dx, row) { return clampf(v + dx * list_get(row, 4), list_get(row, 2),
list_get(row, 3)) }
bind start = do {
  bind node = [["px", 1.5], ["py", -2.0], ["rx", 0.5236], ["sx", 1.0], ["vis", 1.0]]
  bind r = rows()
  bind i = 0
  while i < len(r) {
    bind row = list_get(r, i)
    bind v = list_get(list_get(node, i), 1)
    println(list_get(row, 1), ": ", fmt_value(v, list_get(row, 5)))
    bind i = i + 1
  }
  println("drag Scale X by -500 px: ", fmt_value(drag(1.0, -500.0, list_get(r, 3)),
"num"), " (clamped to the 0.01 minimum)")
  println("drag X by +40 px: ", fmt_value(drag(1.5, 40.0, list_get(r, 0)), "num"))
}
```

```
OUTPUT
X: 1.50
Y: -2.00
Rot X: 30.0 deg
Scale X: 1.00
Visible: 1
drag Scale X by -500 px: 0.01 (clamped to the 0.01 minimum)
drag X by +40 px: 3.50
```

How it works. `fmt_value` formats by row kind (a plain number, an angle in degrees, an integer toggle). `drag` changes a value by `dx * speed` and clamps it to the row's range, which is what `u_dragval` does with the mouse delta; the scale field cannot be dragged below its minimum. Rotation is stored in radians and shown in degrees, so conversion happens only at the edge of the UI.

Variations. Add a row kind for colours (a packed `r * 65536 + g * 256 + b`) or for text fields. Because rows are data, per-node-kind inspectors are different row lists rather than different functions.

Mesh data: flat arrays, edges and normals

LingModeler stores a polygon mesh as plain lists in the store: `vx`, `vy`, `vz` (one number per vertex), `fi` (corner vertex indices of all faces back to back), `fs` (first corner of each face) and `fc` (corner count of each face). That shape supports triangles, quads and n-gons together and copies cheaply. Edges are derived, built lazily and flagged `edirty` when topology changes.

```
# Polygon mesh the way LingModeler stores it: flat coordinate lists + corner index lists.
#   vx vy vz : one number per vertex          fi : corner vertex indices, all faces back to back
#   fs : first corner of each face            fc : corner count of each face (tris, quads, n-gons)
fn cube() {
  return [
    [-1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0],
    [-1.0, -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0],
    [-1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0],
    [0, 3, 2, 1,  4, 5, 6, 7,  0, 1, 5, 4,  2, 3, 7, 6,  0, 4, 7, 3,  1, 2, 6, 5],
    [0, 4, 8, 12, 16, 20],
    [4, 4, 4, 4, 4, 4]
  ]
}
# unique undirected edges: key by "min,max" so a shared edge is stored once
fn build_edges(fi, fs, fc) {
  bind seen = []
  bind ea = []
  bind eb = []
  bind f = 0
  while f < len(fs) {
    bind s = list_get(fs, f)
    bind n = list_get(fc, f)
    bind c = 0
    while c < n {
      bind a = list_get(fi, s + c)
      bind b = list_get(fi, s + (c + 1) % n)
      bind lo = min(a, b)
      bind hi = max(a, b)
      bind key = num_str(lo) + "," + num_str(hi)
      if list_has(seen, key) == false {
        bind seen = list_push(seen, key)
        bind ea = list_push(ea, lo)
        bind eb = list_push(eb, hi)
      }
      bind c = c + 1
```

```
    }
    bind f = f + 1
  }
  return [ea, eb]
}
fn list_has(l, v) {
  bind i = 0
  while i < len(l) {
    if list_get(l, i) == v { return true }
    bind i = i + 1
  }
  return false
}
# Newell's method: normal of a polygon of any size (works for quads and n-gons)
fn face_normal(vx, vy, vz, fi, s, n) {
  bind nx = 0.0
  bind ny = 0.0
  bind nz = 0.0
  bind c = 0
  while c < n {
    bind a = list_get(fi, s + c)
    bind b = list_get(fi, s + (c + 1) % n)
    bind ax = list_get(vx, a)
    bind ay = list_get(vy, a)
    bind az = list_get(vz, a)
    bind bx = list_get(vx, b)
    bind by = list_get(vy, b)
    bind bz = list_get(vz, b)
    bind nx = nx + (ay - by) * (az + bz)
    bind ny = ny + (az - bz) * (ax + bx)
    bind nz = nz + (ax - bx) * (ay + by)
    bind c = c + 1
  }
  bind l = sqrt(nx * nx + ny * ny + nz * nz)
  return [nx / l, ny / l, nz / l]
}
bind start = do {
  bind m = cube()
  bind vx = list_get(m, 0)
```

```
  bind vy = list_get(m, 1)
  bind vz = list_get(m, 2)
  bind fi = list_get(m, 3)
  bind fs = list_get(m, 4)
  bind fc = list_get(m, 5)
  bind e = build_edges(fi, fs, fc)
  println("V=", len(vx), " F=", len(fs), " E=", len(list_get(e, 0)), "  Euler V-E+F =
", len(vx) - len(list_get(e, 0)) + len(fs))
  bind f = 0
  while f < len(fs) {
    bind nrm = face_normal(vx, vy, vz, fi, list_get(fs, f), list_get(fc, f))
    println(format("face {} normal ({:.0}, {:.0}, {:.0})", f, list_get(nrm, 0),
list_get(nrm, 1), list_get(nrm, 2)))
    bind f = f + 1
  }
}
```

```
OUTPUT
V=8 F=6 E=12  Euler V-E+F = 2
face 0 normal (0, 0, -1)
face 1 normal (0, 0, 1)
face 2 normal (0, -1, 0)
face 3 normal (0, 1, 0)
face 4 normal (-1, 0, 0)
face 5 normal (1, 0, 0)
```

How it works. `build_edges` walks every face's corners and stores each undirected edge once, keyed `"min,max"`. A cube has 8 vertices, 12 edges and 6 faces, and Euler's formula V - E + F = 2 is a quick check that a mesh is closed. Newell's method computes the normal of a polygon of any size; all six normals point outward.

Variations. For large meshes replace the string key with `lo * vertex_count + hi` and a sorted list. Face normals feed flat shading; per-vertex normals are the normalised sum of the adjacent faces' normals.

Vector paths: Bezier nodes and flattening

A path in the vector editor is a list of six numbers per node: the anchor, the incoming handle and the outgoing handle, each as `x y`. To draw or hit-test it, the editor flattens the curves into a polyline (`vc_flatten_pts`).

```
# Vector path: bezier nodes are 6 numbers each [x y  in-handle x y  out-handle x y].
# Flatten to a polyline by sampling each cubic segment (adapted from vc_flatten_pts).
fn ellipse_nodes(cx, cy, rx, ry) {
  bind k = 0.5522847
  # right, bottom, left, top; each node's handles are tangent to the ellipse
  return [cx + rx, cy,  cx + rx, cy - ry * k,  cx + rx, cy + ry * k,
          cx, cy + ry,  cx + rx * k, cy + ry,  cx - rx * k, cy + ry,
          cx - rx, cy,  cx - rx, cy + ry * k,  cx - rx, cy - ry * k,
          cx, cy - ry,  cx - rx * k, cy - ry,  cx + rx * k, cy - ry]
}
fn flatten(pts, closed, steps) {
  bind n = floor(len(pts) / 6)
  bind out = []
  bind segs = n - 1
  if closed { bind segs = n }
  bind i = 0
  while i < segs {
    bind j = (i + 1) % n
    # cubic from node i (anchor, out-handle) to node j (in-handle, anchor)
    bind ax = list_get(pts, i * 6)
    bind ay = list_get(pts, i * 6 + 1)
    bind bx = list_get(pts, i * 6 + 4)
    bind by = list_get(pts, i * 6 + 5)
    bind cx = list_get(pts, j * 6 + 2)
    bind cy = list_get(pts, j * 6 + 3)
    bind dx = list_get(pts, j * 6)
    bind dy = list_get(pts, j * 6 + 1)
    bind s = 0
    while s < steps {
      bind t = s / steps
      bind u = 1.0 - t
      bind out = list_push(out, u * u * u * ax + 3.0 * u * u * t * bx + 3.0 * u * t * t
* cx + t * t * t * dx)
      bind out = list_push(out, u * u * u * ay + 3.0 * u * u * t * by + 3.0 * u * t * t
* cy + t * t * t * dy)
      bind s = s + 1
    }
    bind i = i + 1
```

```
  }
  return out
}
fn poly_area(flat) {
  bind a = 0.0
  bind n = len(flat) / 2
  bind i = 0
  while i < n {
    bind j = (i + 1) % n
    bind a = a + list_get(flat, i * 2) * list_get(flat, j * 2 + 1) - list_get(flat, j *
2) * list_get(flat, i * 2 + 1)
    bind i = i + 1
  }
  return abs(a) / 2.0
}
bind start = do {
  bind nodes = ellipse_nodes(0.0, 0.0, 100.0, 100.0)
  bind flat = flatten(nodes, true, 16)
  println("nodes: ", len(nodes) / 6, "  polyline points: ", len(flat) / 2)
  println("area of the flattened circle: ", round(poly_area(flat) * 10.0) / 10.0, "
(pi r^2 = ", round(3.14159265 * 100.0 * 100.0 * 10.0) / 10.0, ")")
  bind coarse = flatten(nodes, true, 4)
  println("with 4 steps per segment the area is ", round(poly_area(coarse) * 10.0) /
10.0)
}
```

```
OUTPUT
nodes: 4  polyline points: 64
area of the flattened circle: 31374.2  (pi r^2 = 31415.9)
with 4 steps per segment the area is 30621.8
```

How it works. The cubic from node `i` to node `j` uses node `i`'s outgoing handle and node `j`'s incoming handle as control points. A circle is four nodes whose handles are 0.5522847 of the radius away (the standard kappa value). Sampling 16 steps per segment gives an area within 0.15 percent of the true circle; with 4 steps, 2.5 percent. Production code chooses the step count from the curve length and the on-screen tolerance (clamped to 2 to 32).

Variations. The shoelace formula in `poly_area` also gives the winding direction (drop the `abs`). Hit-testing a filled path is a point-in-polygon test on the flattened points.

Procedural textures

LingModeler's texture lab has 16 generators (noise, cells, bricks, wood, marble, fire, and more) built on a handful of functions: a hash, value noise, and fractal Brownian motion (fBm), which sums noise at doubling frequencies and halving amplitudes. The preview here is text; in the program the same function fills a pixel buffer.

```
# Procedural texture lab: hash noise -> value noise -> fBm -> tileable variant,
previewed as text.
fn hash3(x, y, z, seed) {
  bind v = sin(x * 127.1 + y * 311.7 + z * 74.7 + seed * 53.3) * 43758.5453
  return v - floor(v)
}
fn lerp(a, b, t) { return a + (b - a) * t }
# value noise; px/py > 0 wraps the lattice so the result tiles every px x py cells
fn vnoise(x, y, seed, px, py) {
  bind ix = floor(x)
  bind iy = floor(y)
  bind fx = x - ix
  bind fy = y - iy
  bind ux = fx * fx * (3.0 - 2.0 * fx)
  bind uy = fy * fy * (3.0 - 2.0 * fy)
  bind x0 = ix
  bind x1 = ix + 1
  bind y0 = iy
  bind y1 = iy + 1
  if px > 0 { bind x0 = x0 - floor(x0 / px) * px
    bind x1 = x1 - floor(x1 / px) * px }
  if py > 0 { bind y0 = y0 - floor(y0 / py) * py
    bind y1 = y1 - floor(y1 / py) * py }
  return lerp(lerp(hash3(x0, y0, 0, seed), hash3(x1, y0, 0, seed), ux), lerp(hash3(x0,
y1, 0, seed), hash3(x1, y1, 0, seed), ux), uy)
}
fn fbm(u, v, cells, octaves, seed) {
  bind sum = 0.0
  bind amp = 0.5
  bind norm = 0.0
  bind f = 1
  bind i = 0
  while i < octaves {
    bind sum = sum + vnoise(u * cells * f, v * cells * f, seed + i * 7, cells * f,
cells * f) * amp
    bind norm = norm + amp
    bind amp = amp * 0.5
    bind f = f * 2
```

```
    bind i = i + 1
  }
  return sum / norm
}
fn shade(v) {
  bind ramp = " .:-=+*#%@"
  bind i = floor(v * 9.999)
  return substr(ramp, i, 1)
}
bind start = do {
  bind size = 24
  bind y = 0
  while y < size / 2 {
    bind line = ""
    bind x = 0
    while x < size * 2 {
      bind line = line + shade(fbm(x / (size * 2.0), y / (size / 2.0), 4, 3, 11))
      bind x = x + 1
    }
    println(line)
    bind y = y + 1
  }
  println("seam check: u=0 vs u=1 differ by ", round(abs(fbm(0.0, 0.3, 4, 3, 11) -
fbm(1.0, 0.3, 4, 3, 11)) * 1000000.0) / 1000000.0)
}
```

```
OUTPUT
::::::::::----:::----------==+++++++++++++==---:
::::::::--========--========++++++++++++++==--:::
...::::::::::---===++++++++++++++==========---::.
::::::::.....::----===++++++++++========--:::::..
:::....::::::-------======++++++====--------::::
-::::------=======--::::::::--=======================:::
..:-==++========--:::::::...:::----::---====----:.
...::---===---====----------=====--------==--::.
..::::----====++++++========+++++=-----=-----::.
--:::::-=++++++++++++==++++++++++++++==-:::::::-
------===++===+++++++=+++++++++++++++======--:::--
-----======--=======-=====----===++++++++++++==-----
seam check: u=0 vs u=1 differ by 0
```

How it works. `hash3` is the classic `fract(sin(dot) * 43758.5453)` pseudo-random function. `vnoise` interpolates the hash at the four lattice corners with a smoothstep curve. Passing `px` and `py` wraps the lattice coordinates (`x - floor(x / px) * px`), which makes the texture **tile seamlessly**: the last line shows that column 0 and column 1 of the tile are identical. `fbm` normalises by the total amplitude so the output stays within 0 to 1.

Variations. Take `abs(noise - 0.5)` for marble veins, or distort the lookup with a second noise for wood. Add a `t` argument as a third coordinate to animate (the fire, water and plasma textures do this). Map the value through a three-stop colour ramp.

Autosave with a dirty flag and an atomic write

Autosave must be cheap when nothing changed, must not stall the frame, and must never leave a half-written file. LingModeler's `autosave_tick` is called every frame; it checks a timer, then the `dirty` flag, then writes `autosave.lmesh`. The write below uses a temporary file and a rename.

```
# Autosave: a dirty flag, a timer, and an atomic write (temp file, then rename) so a crash
# can never leave half a file. Adapted from LingModeler's autosave_tick.
fn mark_dirty() { st_set("dirty", 1.0) }
fn save_atomic(path, text) {
  bind tmp = path + ".tmp"
  write_file(tmp, text)
  if file_exists(path) { remove_file(path) }
  return rename_file(tmp, path)
}
# call once per frame with the clock; saves at most once every `mins` minutes, and only when something changed
fn autosave_tick(now, mins, path) {
  if mins < 0.5 { return "off" }
  if st_has("autosave_t") < 0.5 { st_set("autosave_t", now) }
  if now - st_get("autosave_t", now) > mins * 60.0 {
    st_set("autosave_t", now)
    if st_get("dirty", 0.0) > 0.5 {
      save_atomic(path, st_get("doc", ""))
      st_set("dirty", 0.0)
      return "autosaved"
    }
    return "clean, skipped"
  }
  return "waiting"
}
bind start = do {
  bind path = "autosave.txt"
  st_set("doc", "version 1")
  mark_dirty()
  # a fake clock: 0 s, 100 s, 200 s, 400 s ...
  bind clock = [0.0, 100.0, 200.0, 400.0, 500.0, 700.0]
  bind i = 0
  while i < len(clock) {
    if i == 4 { st_set("doc", "version 2")
      mark_dirty() }
    println("t=", list_get(clock, i), " s -> ", autosave_tick(list_get(clock, i), 3.0, path))
```

```
    bind i = i + 1
  }
  println("on disk: ", read_file(path), "   leftover temp file: ", file_exists(path +
".tmp"))
  remove_file(path)
}
```

```
OUTPUT
t=0 s -> waiting
t=100 s -> waiting
t=200 s -> autosaved
t=400 s -> clean, skipped
t=500 s -> waiting
t=700 s -> autosaved
on disk: version 2   leftover temp file: false
```

How it works. The fake clock shows the schedule: the timer starts at the first call; at 200 s (more than three minutes) a dirty document is saved; at 400 s nothing changed, so the check is skipped; after the edit at 500 s the next window saves again. `write_file` to `path.tmp`, then `rename_file`, means a crash leaves either the old file or the new one. In the real program the interval is the `autosave` setting in minutes (default 5, 0 disables it).

Variations. Autosave to a separate file, never over the user's file, and offer it on the next start if it is newer. LingStationX also keeps `autosave_untitled.json` and a `recover.txt` for the same reason. Use `st_get("now")`, the frame's cached clock, instead of calling `time_now()` repeatedly.

PART

Showcase

Real applications written entirely in Ling.

CHAPTER 16

Case Studies: Applications Built in Ling

These are real programs, shipped by the author of the language, written in Ling from the first line to the last. They are included to show what the language does at scale and what its authors learned doing it. Line counts are lines of Ling in the application's own `ling/` folder, measured with `wc -l`, and exclude assets, generated data and the Rust crates the programs call. Screenshots were taken from the running programs; the wallet's were taken with its simulated (`--mock`) backend, so every balance, address and peer on them is made up.

Application	Kind	Ling code	Native help
LingStationX	Digital audio workstation	25,900 lines, 60 modules	`ling-daw` audio engine (Rust)
Ling Wallet	Post-quantum wallet, node and miner	8,800 lines, 33 modules	`ling-crypto`, `ling-net` (Rust)
LingDev	Game scene, timeline and cutscene editor	28,400 lines, 52 modules	Software 3D renderer is Ling
LingModeler	3D modeler, vector studio, texture lab	16,000 lines, 32 modules	None beyond the runtime
Horizon	Web browser with its own HTML, CSS and script engines	16,300 lines, 12 modules	Image and video decoders
Soul Symphony, Ling Edition	3D game in six languages	26,000 lines of game code	Runtime graphics and audio

LingStationX

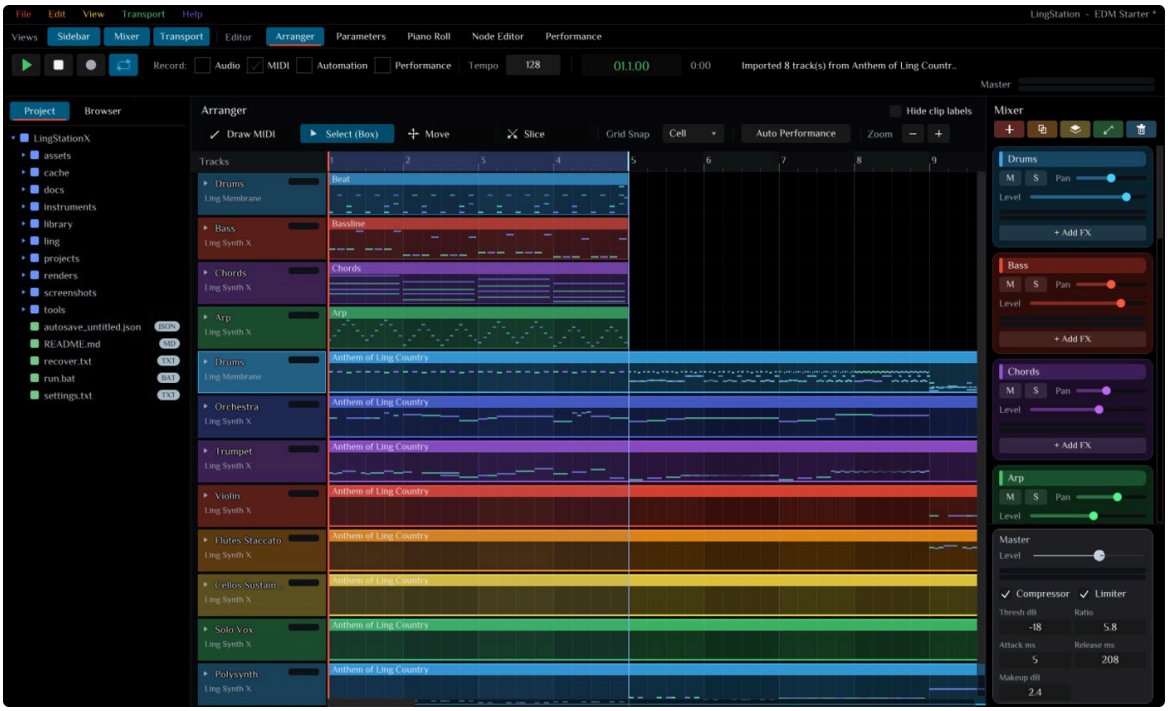


Figure 90.1. The arranger. Each track tile, clip, ruler tick and mixer strip is drawn by Ling code every frame; the clips of an imported eight-track song sit below four template tracks.

What it is. A complete digital audio workstation: arranger with automation lanes, piano roll with per-note expression, mixer with effect racks, a node editor for signal routing, a clip launcher for live performance, a file browser, a theory assistant, MIDI and audio import and export, and several instruments (a subtractive, additive and FM synth, a physical-modelling drum machine, a bass synth, a 16-voice synth called Prism, and a formant singer). It ships for Windows, Linux and macOS, and as an app inside the LingOS operating system on x86-64 and Raspberry Pi 5.

Why Ling fits. A DAW has two halves with opposite needs: a hard-real-time audio engine and a huge, fast-changing user interface. Ling splits them cleanly. The engine is a Rust crate that Ling controls through `daw_*` builtins, so the audio thread never waits on a script. Everything the user touches is Ling: a screen is a function, a widget is a function, and a new panel is added in an afternoon without touching the engine. Instruments are text patches, and users can write their own effects and instruments in a few lines of the LingDSP expression language in `scripts/`, reloaded while the program runs.

Size and structure. 60 modules in `ling/` (25,894 lines): `core` for input and widgets, `ds` for the design system, `actions` for the `do_action(id)` command dispatcher, then one module per screen (`arranger`, `roll`, `mixer`, `nodes`, `perf`). More than 900 patch files and several project templates live in `instruments/` and `assets/`. Chapter 10 walks through the architecture.



Figure 90.2. The Parameters tab showing Ling Prism, a 16-voice synthesiser, with its oscillators, filter, envelopes and modulation matrix. The panel is a Ling function reading the instrument's parameter list from the engine.

Lessons learned.

1. **Put the engine behind a narrow builtin API and keep policy in the script.** The Rust side offers verbs (`daw_note_add`, `daw_undo_push`); the Ling side decides when to call them. It made the engine testable headless, and the same `daw_*` calls render previews on a web server.
2. **Immediate-mode UI works if you cache what is expensive.** Redrawing everything every frame is simple and correct, but a per-note drawing loop in script cost about 2 ms per clip per frame. The fix was one native call (`px_notes`) and a cache keyed on the project revision.
3. **Shared state belongs in one store.** A function cannot see its caller's variables, so the program keeps all state in `st_get` / `st_set`, prefixed by module. It reads like global variables but is searchable, serialisable, and drivable from tests.

4. **Make the program testable from the command line early.** `--shot`, `--do` and `--drive` replay scripted input and write a screenshot. `tools/smoke.py` opens every dialog, tab and menu this way, and the project's `screenshots/` folder holds more than a thousand captures.
5. **Know the language's sharp edges and write them down.** The team's `DEV_NOTES.md` lists them: every keyword of every supported human language is reserved (so `si`, `ok` and `fit` cannot be variable names), `&&` does not short-circuit so bounds must be checked first, and boolean-returning builtins differ from number-returning ones.

Ling Wallet

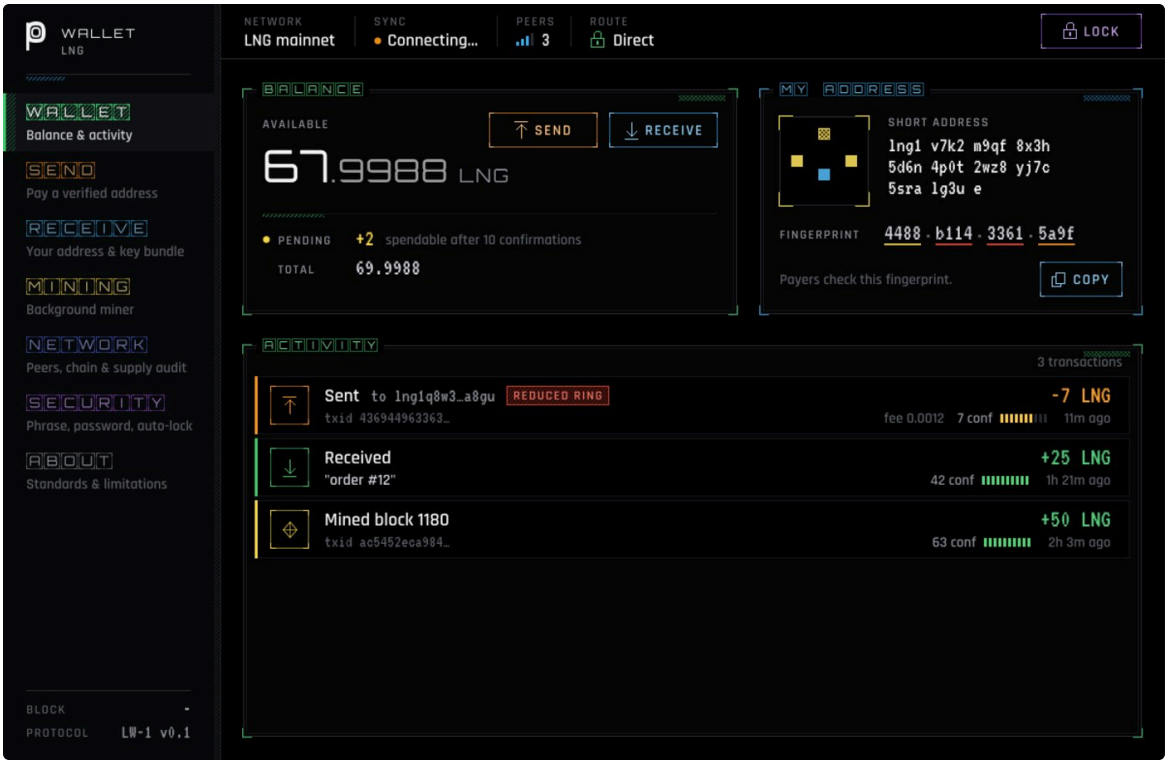


Figure 90.3. The wallet screen on the simulated backend: balance, the fingerprint of the user's address and three activity rows, including a send flagged as using a reduced ring.

What it is. A desktop wallet, full node and background miner for LNG, an experimental privacy coin, in one program. Payments use fresh one-time addresses, ring signatures over same-value decoys, a hybrid post-quantum key exchange, and Dandelion++ propagation. The

wallet file is encrypted with Argon2id and XChaCha20-Poly1305. A 4-D "HyperTorus" visualisation animates the miner. It is explicitly labelled unaudited and experimental, in its README, its About screen and a written threat model.

Why Ling fits. The project's thesis is that a security-critical program is easier to trust when it is all in one readable language. Ling contributes three things that matter here: **isolates with capabilities** (the miner can only hash; only the wallet thread can create a key), **opaque key handles** (secrets never become Ling values), and a **strict byte layer** (hex strings plus bounds-checked readers). The UI and the engine meet at a small contract of non-blocking `be_*` calls, implemented once for the real node and once as a mock.

Size and structure. 33 modules (8,824 lines): `core/` for consensus and wallet logic without I/O, `isolates/` for the node, wallet and miner threads, `ui/` for sixteen screen and widget modules, plus `tests/` for byte-compatibility against a reference implementation. Chapter 11 walks through the cryptographic recipes.

Lessons learned.

1. **Make the architecture enforce the security policy.** Capabilities are checked by the runtime, not by convention. A reviewer can answer "can the miner read the wallet file?" from one `thread_spawn` line.
2. **Ship a mock backend.** `--mock` made every screen designable and screenshot-testable without a chain, a network or a real key. It also stopped screenshots from ever containing real data.
3. **Test against a reference.** Expected values in `t_crypto.ling` came from the original Rust reference implementation, so a passing test means byte compatibility, not merely self-consistency.
4. **Put the checks where a human can fail.** The send flow shows the recipient's fingerprint and refuses to continue until the user confirms it matches, because a checksum protects against typos but not against a swapped address.
5. **Be honest in the product.** Standard, novel and experimental primitives are labelled as such on the About screen, and `wipe` is documented as best effort because it is. Trust comes from stating the limits.

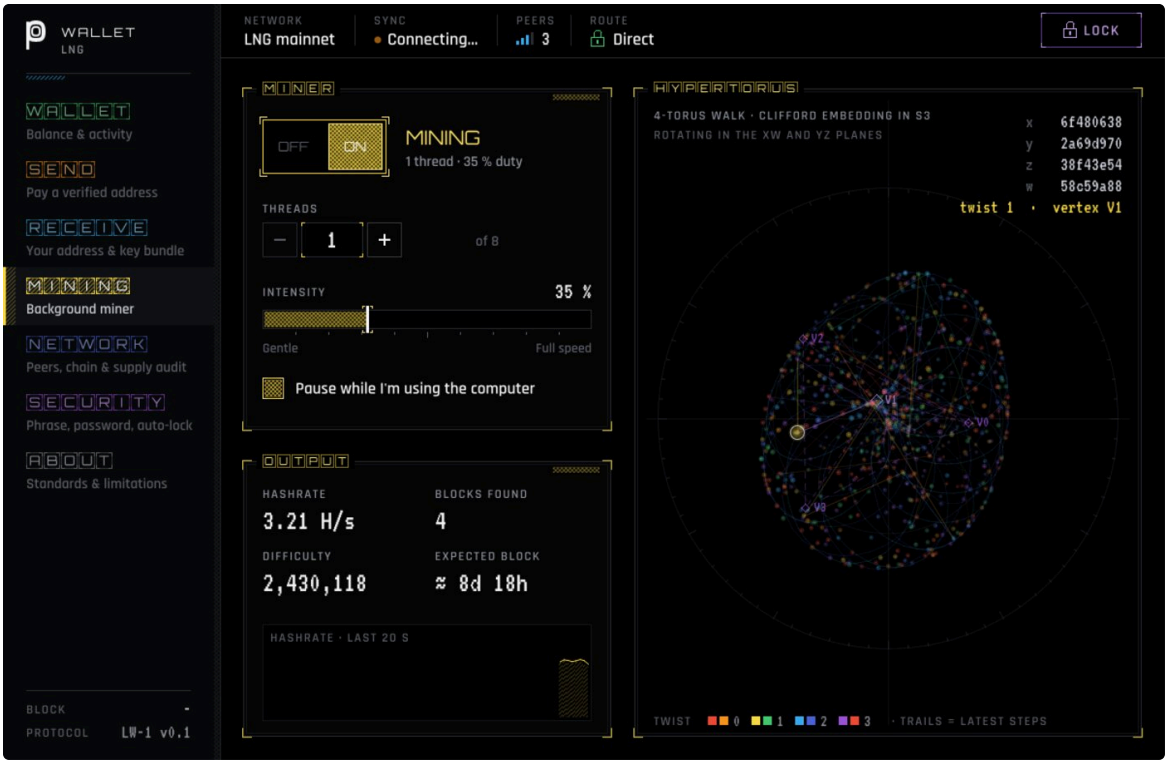


Figure 90.4. The Mining screen on simulated data: thread count, duty-cycle intensity, a pause-while-I-am-using-the-computer option and the 4-D torus walk drawn from `lw_pow_trace` samples.

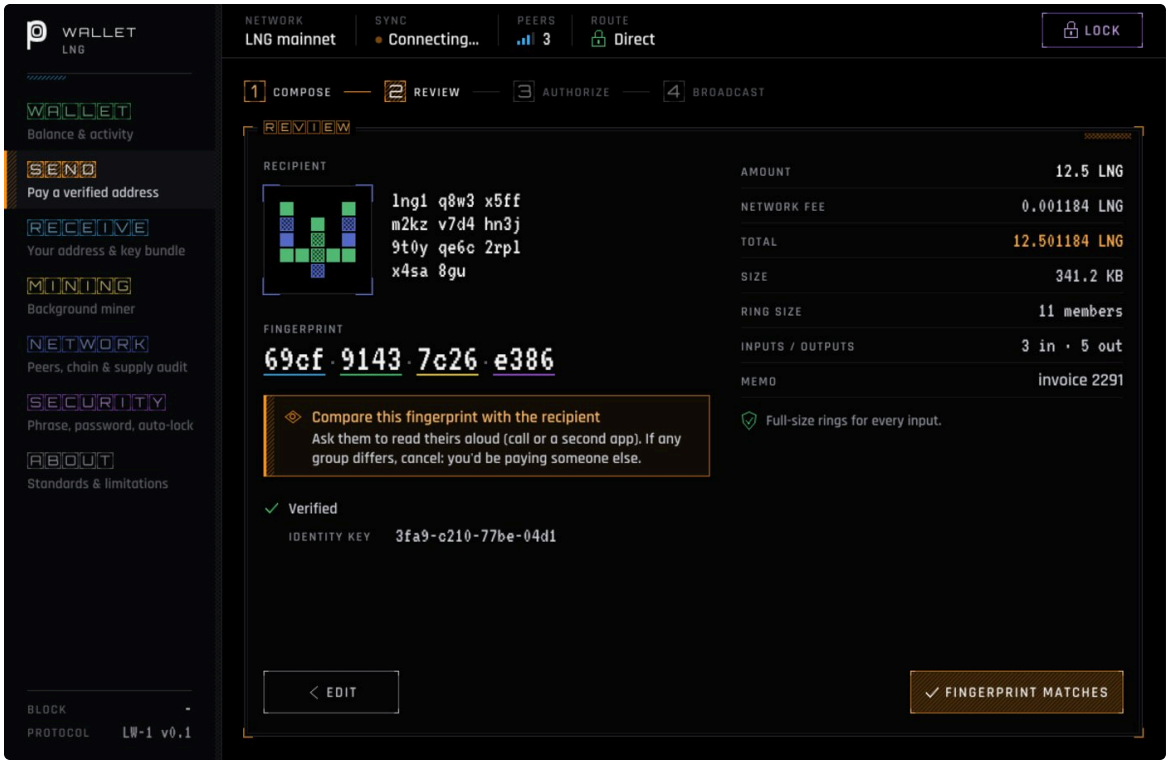


Figure 90.5. The review step of a payment (simulated data): the recipient's identicon and fingerprint, fee, ring size and a prompt to compare the fingerprint out of band.

LingDev

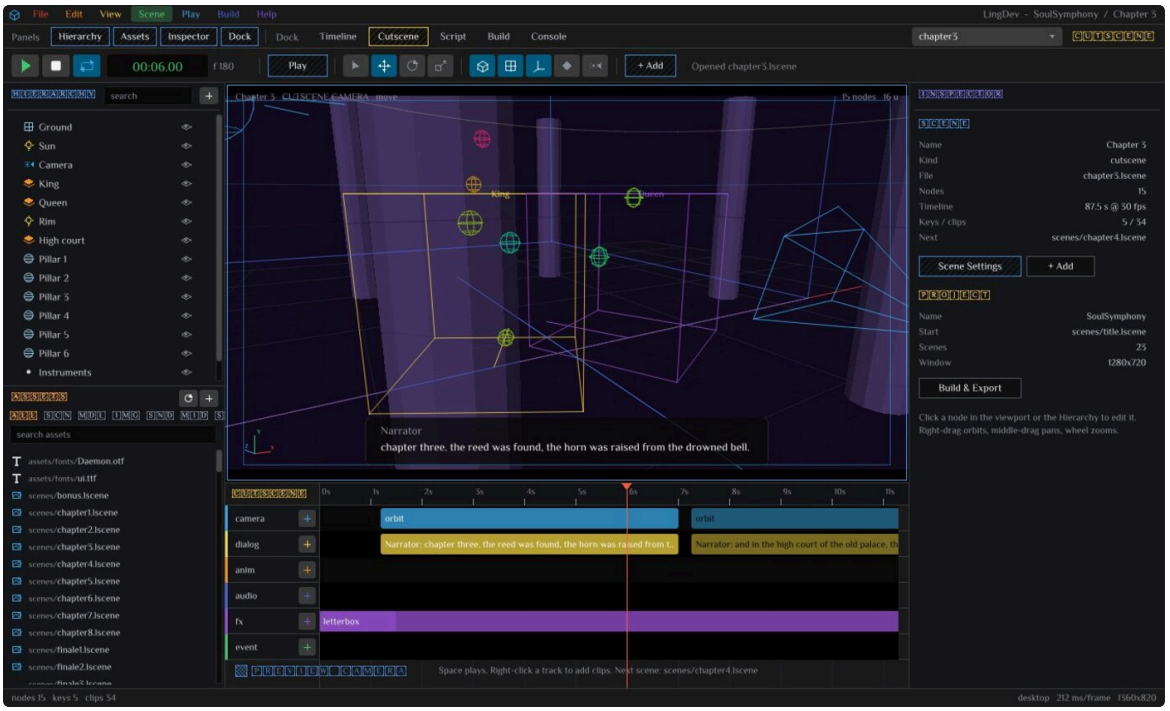


Figure 90.6. LingDev editing a cutscene of the Soul Symphony Ling edition: hierarchy and assets on the left, the 3D viewport with the camera path, the inspector on the right and the cutscene tracks in the dock.

What it is. A scene, timeline and cutscene editor for games, itself written in Ling. It edits 3D levels, 2D menus and HUDs, and cinematic cutscenes with six track types (camera, dialog, animation, audio, effects, events). It has a viewport with gizmos, a Flash-style keyframe timeline with 21 easing curves, a script editor, a one-key build panel for Windows, Linux, macOS, the web, a Raspberry Pi kernel image and LingOS, and importers that turned existing games into projects. Six complete games are built with it: a first-person shooter, a breakout game, a rhythm game, an arena shooter, a storybook platformer and the 2019 Soul Symphony.

Why Ling fits. An editor and the thing it edits should speak the same language. LingDev's runtime (19 modules, 18,500 lines) is Ling, copied into every project, and the editor `use`s the same modules to play a game inside its own window. There is no export step that can disagree with the preview.

Size and structure. 28,355 lines: 9,856 in the editor (`app`, `project`, `timeline`, `cutscene`, `inspector`, `hier`, `view3d`, `view2d`, `buildpanel`, `importers`) and 18,499 in the shared runtime `rt/`. Projects are a `lingdev.toml`, line-based `.lscene` files and plain `.ling` scripts. Chapter

12 has recipes from it.

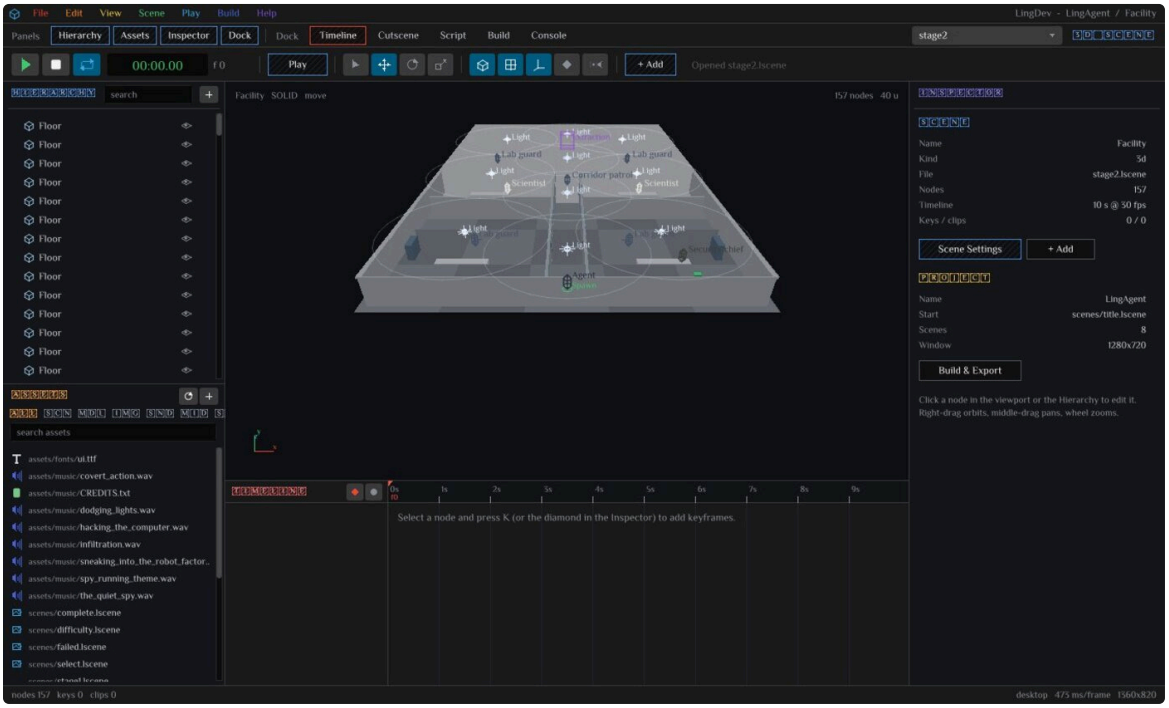


Figure 90.7. The same editor editing a first-person stage: guards, lights and the extraction point in the 3D viewport, drawn by the software renderer in the shared runtime.

Lessons learned.

1. **One runtime for the editor and the game.** Parsing, serialising and rendering a scene happen in shared code, so the preview cannot drift from the product.
2. **Design files for diffs.** Line-per-record text with ignorable unknown fields lets artists use version control and lets old projects open in new versions.
3. **Snapshots beat inverse operations for undo.** Serialising the scene to a string takes a few lines and makes every edit undoable, including edits that did not exist yet.
4. **A software renderer is a feature.** Drawing 3D with Ling code (`rt_sw3d`) means the same viewport code runs on the desktop and in the LingOS builds, without depending on a GPU API.
5. **Easing and interpolation are worth writing in the language.** `rt_ease.ling` runs identically on every platform, and a designer can read it.

LingModeler

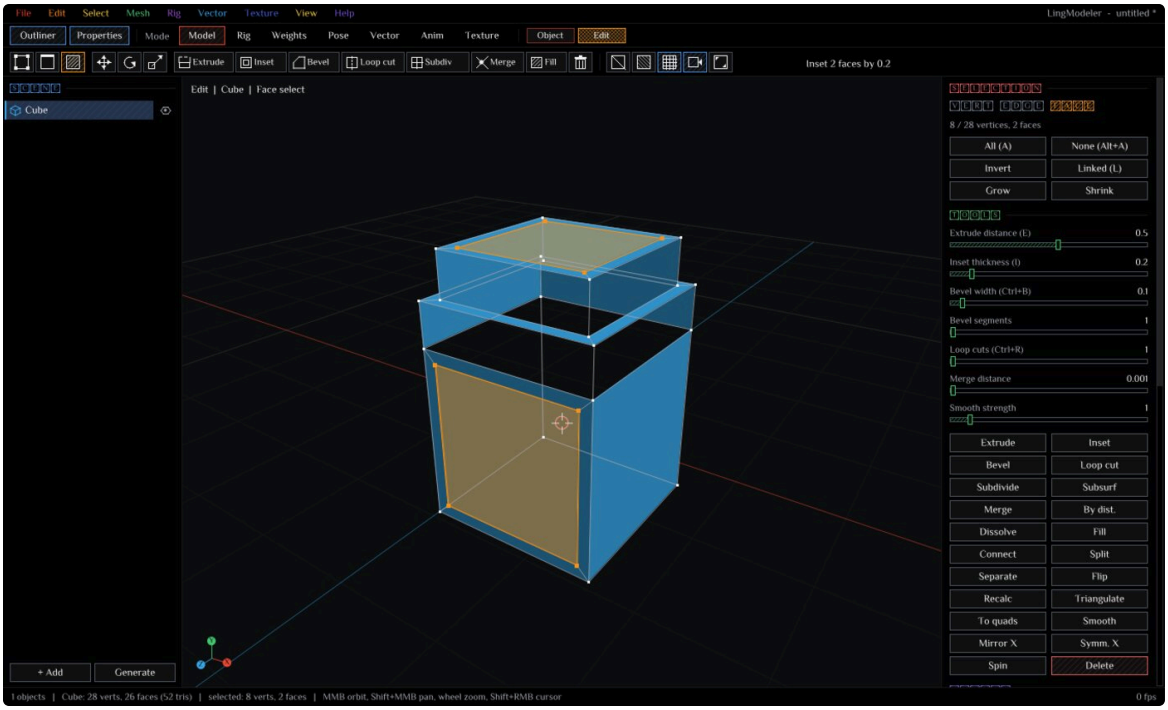


Figure 90.8. Edit mode on a box mesh, with the extrude and inset panels on the right.

What it is. A lightweight 3D modeler, 2D vector illustrator, animation studio, procedural texture lab and rigging tool in one window. Modes cover object and edit (box modeling with extrude, bevel, loop cut, subdivision, UV projections), rig (a 19-bone character armature), weight painting with auto weights, pose and keyframed actions, vector paths and shapes, 2D animation with onion skin, and 16 procedural texture generators. It reads and writes OBJ, glTF, SVG and PNG, and its native `.lmesh` format is valid Ling source.

Why Ling fits. Ling's lists and string-keyed store make a mesh a handful of flat arrays; copying them for undo is cheap because values share storage until edited. Because the project file is itself Ling source, a game can `use "hero.lmesh"` and call `draw_Hero(...)` without writing a loader. The same program runs on Windows, macOS, Linux, and inside LingOS on x86-64 and Raspberry Pi 5.

Size and structure. 32 modules (16,015 lines). The largest are `meshops` (1,873, the modeling operators), `vec2d` (1,537), `view3d` (1,534) and `rig` (1,205).

Lessons learned.

1. **Choose data layouts that fit the language.** Flat parallel lists (`vx`, `vy`, `vz`, `fi`, `fs`, `fc`) are simple, copy cheaply and handle tris, quads and n-gons together.
2. **A file format that is also source code is free documentation.** Anyone can open a `.lmesh` and read it.
3. **Keep a procedural texture engine tiny.** Sixteen generators are a hash, a noise, an fBm and some arithmetic. Pass the tile period into the noise and every texture can be seamless.
4. **State limits plainly.** The README lists what the importers skip (SVG text, gradients, clip paths; glTF animations). Users forgive limits they were told about.

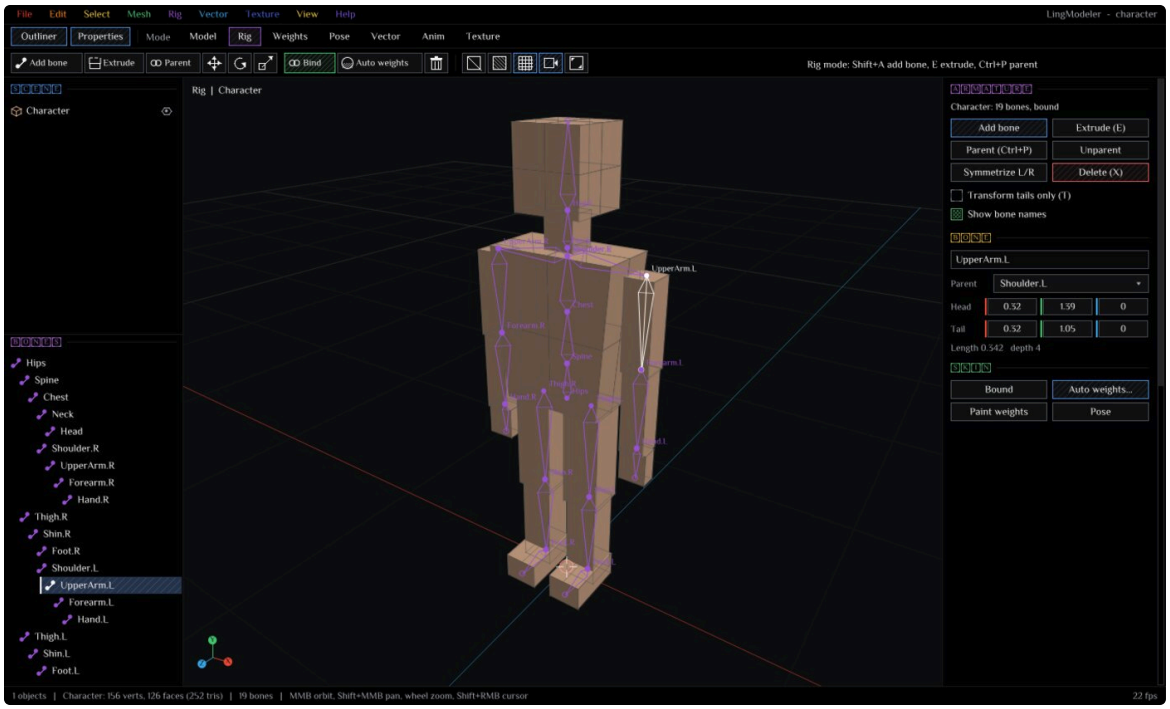


Figure 90.9. Rig mode: a generated character with its 19-bone armature, bone hierarchy on the left and bone properties on the right.

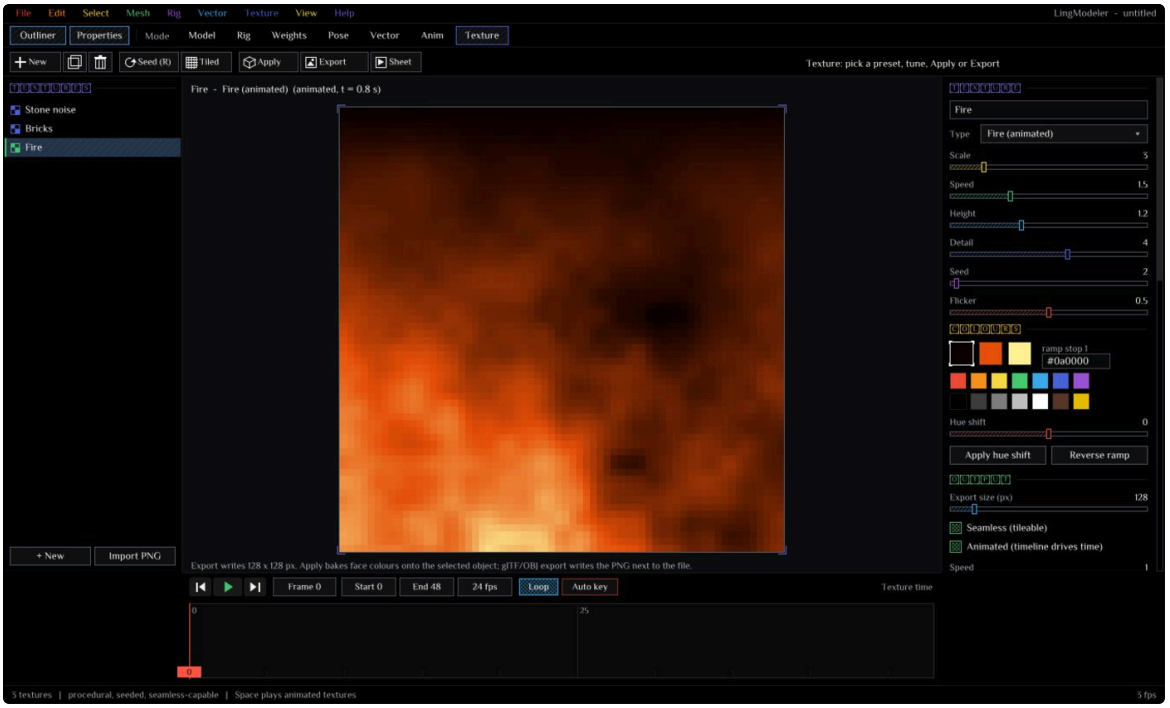


Figure 90.10. The texture lab showing the animated fire generator with its colour ramp and seamless-tiling options.

Horizon

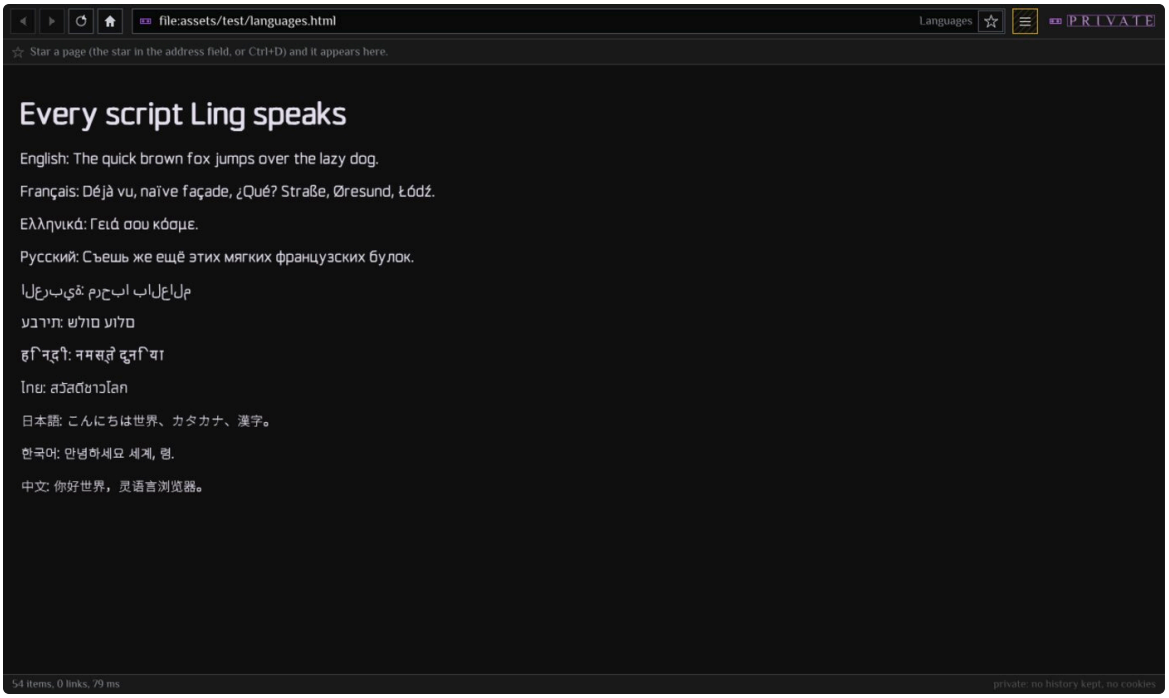


Figure 90.11. Horizon rendering its test page `languages.html`: Latin, Greek, Cyrillic, Arabic, Hebrew, Devanagari, Thai, Japanese, Korean and Chinese, set in its own stroke font.

What it is. A private web browser, a Ling program in the look of the other apps: toolbar, address field, bookmarks bar, status line. It parses HTML and CSS (selectors, cascade, flexbox, grid, tables, floats, media queries), lays pages out, interprets page scripts in a sandbox it implements itself (no JavaScript engine is involved), draws images and plays H.264 video through the runtime's player, and removes tracking parameters. History is kept only in memory, with no cache and no cookies. It also exists as a `no_std` Rust layout library for LingOS.

Why Ling fits. A browser is mostly text processing and tree walking, which Ling does well, and the headline lesson is that it is fast enough. Rules are bucketed by their rightmost class, id or tag, with ancestor filters, which makes a sheet of 20,000 rules workable. A headless mode (`--dump file:page.html`) prints the laid-out page, so the engine is tested without a window; the language test page runs 83 checks of the script interpreter and reports "all 83 checks passed" in about 280 ms.

Size and structure. 16,264 lines in `ling/`: `html`, `css` (3,250), `layout` (3,295), `jsdom` (3,964), `js` and `jsrun`, `ui` and `app`. Image and video decoding, networking and fonts are runtime builtins.

Lessons learned.

1. **Write a headless mode first.** `--dump` and `--click` made the engine testable and let a document check itself.
2. **Budget everything a page can do.** Scripts get a time allowance of a few seconds at load, 0.4 seconds per event and 60 ms per animation frame; over that, a script is stopped and the page carries on.
3. **Block by default.** `fetch`, cookies, `eval` and workers are off on purpose. The privacy features are the absence of features.
4. **Know your backend.** The browser's notes say to run it on the default JIT, because the tree-walking interpreter runs out of stack at about 40 nested calls on Windows, too few for the script sandbox. Other apps in this chapter are developed on the interpreter. Test your program on every backend you ship.

Soul Symphony, Ling Edition

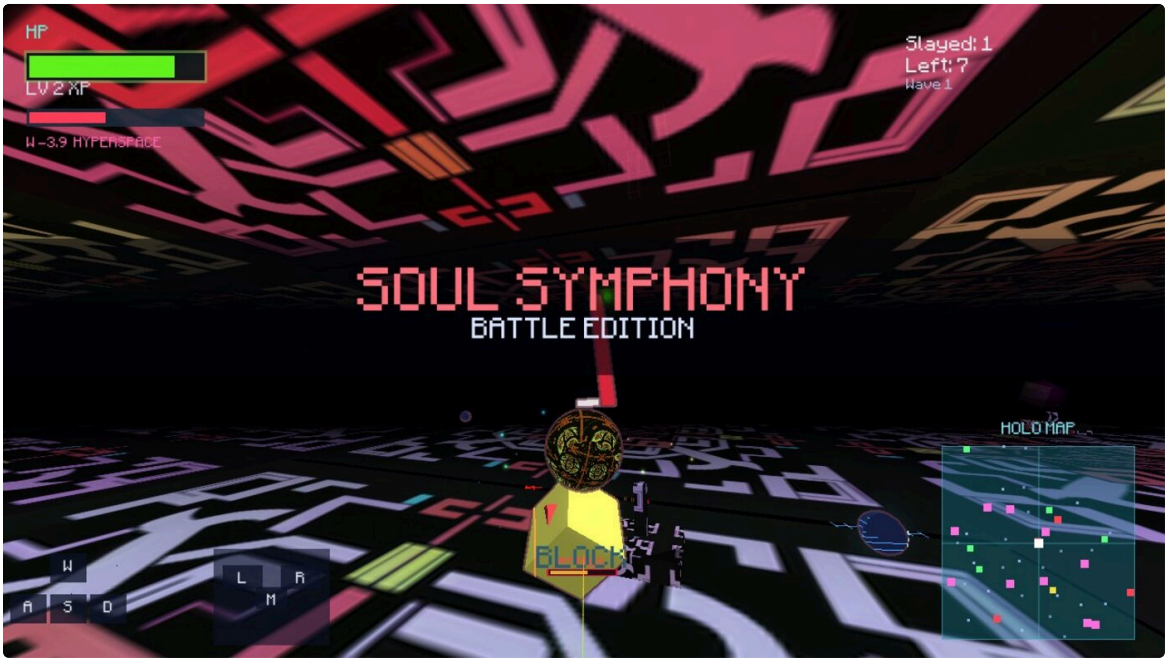


Figure 90.12. Soul Symphony Battle Edition running from a LingDev project: vector lattices, a heads-up display and a holographic map.

What it is. A spiritual 3D adventure in a soul realm: collect the letters of the world-song, clear a monster lattice and duel in an arcade arena. It is a port of the author's 2019 game with full voice-over and subtitles in six languages (1,578 lines of voice-over), and it is released for Windows, Linux and macOS.

Why Ling fits. The project describes itself as written end to end in Ling. Geometry is vector code, levels and bosses are Ling modules, and dialogue is stored per language as Ling data modules. LingDev can import the game's cutscenes, voice-over, models, music and hub level, which turns a finished game back into an editable project.

Size and structure. 47 files and about 26,000 lines in `game/` (arcade, arena, boss, companions, level, lattice, audio). The repository has more than 650 `.ling` files in all; most of the additional lines are dialogue font data for Chinese, Japanese, Korean and Thai, produced by tooling, not typed.

Lessons learned.

1. **Separate content from code.** Chapters are previews (`chapter1_preview.ling` through `chapter8_preview.ling`), so each can be tested alone.
2. **Treat localisation as data.** Six languages of text, voice and fonts are data modules selected at run time.
3. **Make a finished game importable.** Because the game is data and Ling modules, an editor can open it, which is the strongest test of an open format.

Also shipped

Smaller programs from the same workshop show Ling on the server side. They are described in their READMEs; the line counts are lines of Ling in each project.

Project	What it is	Lines of Ling
`studio.linglin.art`	A web site that sells LingStationX, hosts subscribers' cloud projects, and renders previews offline with the DAW engine (`daw_bounce`) in detached jobs; SQLite storage	about 9,100 in `app`, `controllers` and `models`
`jadepos.store`	A point-of-sale for merchants on `ling-http` and `ling-crypto`: Argon2id, BLAKE3, TOTP two-factor accounts, card validation, and only brand and last four digits stored	9,310
`fu.ling-lang.org`	The package registry for Ling, like crates.io, with two-factor accounts, written in Ling (Thai keywords)	5,620

Project	What it is	Lines of Ling
`countries.ling-gov.org`, `music.ling-lin.me`	Small site back ends in the same house style	3,594 and 1,982

What the larger programs have in common is worth stating once more. They keep a thin native layer (an audio engine, a crypto library, decoders) and write everything else, including user interfaces, file formats, parsers, renderers and test harnesses, in Ling. They share one visual design system and one set of conventions, so a skill learned in one transfers to the others. And a skill learned in one of them, such as reading a `.lmesh` or a `.lscene`, is plain Ling, so it transfers without a toolchain.

APPENDIX

Known Limitations and Gotchas

Ling is a young language that moves quickly. This appendix collects the behaviours that surprise newcomers most often, so that you can recognise them in minutes instead of hours. Everything here was verified against Ling 2030.1.53 using both backends: the default Cranelift JIT (`ling run`) and the tree-walking interpreter (`ling run --interp`).

> TIP
>
> When a program misbehaves, run it once with `ling run --interp`. The interpreter reports unknown functions and runtime errors that the JIT silently turns into `()`.

Unknown names fail silently on the JIT

On the default backend, calling a function that does not exist returns the unit value `()` and the program carries on. A typo such as `draw_rect` (which does not exist) produces no error and no drawing. The interpreter reports `unknown function`. Make `--interp` part of your edit-run loop whenever something "does nothing".

There is no assignment statement

`x = x + 1` is a parse error. Ling rebinds instead: `bind x = x + 1`. Rebinding shadows the earlier binding in the current scope.

Loops and scope

Construct	Does `bind` inside update an outer variable?
`while` body	Yes. The body shares the enclosing scope.
`if` body	Yes.
`for` body	No. Each iteration gets a fresh copy of the scope.

To accumulate a total, use a `while` loop with a counter. A `bind` inside a `do { }` block is discarded on the interpreter but leaks into the outer scope on the JIT, so do not depend on either behaviour.

Range bounds must be simple

`0..n` and `0..(n + 1)` work. `0..n + 1` is a parse error, and `0..len(xs)` silently misbehaves. Wrap the bound in parentheses: `0..(len(xs))`.

Numbers and text

- Numeric literals are decimal digits with at most one fraction. There is no `1e3`, `0xff`, `1_000`, `.5` or `3.`.
- Strings use double quotes only and have no interpolation. Build text with `+` and `str()`.
- `print` and `println` both end the line, and both join their arguments with no separator: `println("sum", 1 + 2)` prints `sum3`.
- Single-precision results can print with noise (`0.800000011920929`). Use `round(x * 100) / 100` when you display them.

Comparing structured values

`==` on lists, records and variants is always false, even for identical contents. Compare fields or elements yourself.

What the type system does and does not do

Ling is dynamically typed. Type annotations such as `x: number` and `-> T` are parsed and ignored. There is no map or dictionary type, no set, no tuple and no JSON parser. Model key-value data with parallel lists or with records, or store it in SQLite through the `db_*` builtins.

Keywords that currently do nothing

These words are reserved by the lexer but have no effect at run time:

`stop`, `again`, `change`, `post`, `give`, `fit`, `can`, `sure`, `maybe`, `pure`, `spawn`, `none`

In particular `stop` and `again` do not break out of or continue a loop. Restructure the loop condition instead. The ownership words `own`, `lend`, `share`, `move` and `copy` are parsed and ignored, with no borrow or move checking. `async` runs synchronously.

Closures

A closure passed to a builtin (`try`, `list_map`, `list_filter` and similar) works only on the interpreter. Closures held in variables are unreliable on the JIT. A named top-level `fn` works everywhere, so prefer it. The one exception is `try()`: on the JIT it always returns an error (`try: first argument must be a function`), even with a named function, and it works only under `--interp`. Programs that call `http_serve` run on the interpreter automatically, so `try` works inside request handlers.

JIT and interpreter differences at a glance

Situation	JIT (default)	Interpreter
Unknown function	returns `()` silently	error
`1 / 0`	`()`	`inf`
`pi` as a bare name	`()` (use `pi()`)	value
`list_push`	may mutate the original list	returns a new list
Empty text or list in a condition	truthy	falsy
Out-of-range index	empty value	error

Always write `bind xs = list_push(xs, v)` so that your code is correct on both.

Tooling

- `lingfu new` creates `src/start.ling`, but `lingfu build` expects `main.ling` unless the manifest says otherwise. Add `entry = "src/start.ling"` to the `[spirit]` section of `Ling.toml`.
- `lingfu check`, `lingfu fmt` and `lingfu translate` currently call `ling` subcommands that do not exist, so they report success without doing anything.
- Localised `lingfu` command names (for example `灵符 运行`) work only when the executable itself is renamed.

- Native `ling build` compiles generated Rust through Cargo and needs a Rust toolchain on the build machine.

Modules and shared code

`use "file"` flattens the loaded file into one namespace. Functions and (in the tested version) top-level `bind` constants of the loaded file are visible, but if two files define the same function name, the last one loaded silently wins, even for callers in the earlier file. Calls between sibling functions inside a `mod` block are unreliable. Prefix your function names with the library name, for example `geom_area`.

Testing

`lingfu test` runs programs on the JIT, which never fails at run time, so it can print "All tests passed!" for a program whose assertions failed. Make your tests print a clear PASS or FAIL line, and check that line, or run them under `ling run --interp`, which stops with an error. Helper files placed inside `tests/` are treated as tests and fail with "no entry point".

Performance

The JIT is more than a hundred times faster than the interpreter for arithmetic such as recursive Fibonacci, but appending to a string in a loop is quadratic on the JIT: 40,000 appends took 1.1 s, against 39 ms on the interpreter. Collect pieces in a list and join them once. Isolates run on the interpreter, so they help with blocking work but not with raw arithmetic.

Parse traps

A line that begins with `[` or `-` continues the previous expression. `wait` is reserved, and `append_file`, `panic` and `exit` are not builtins.

Graphics and audio

- Window coordinates have +y pointing down. A cone's apex points down by default.

- In `font_text`, `font_text_fill` and `ui_text`, `y` is the top of the text line, not its baseline.
- 3D calls are queued and always draw on top of 2D calls unless you call `flush_3d()`.
- Colour arguments are not clamped; 300 spills into the next channel.
- `rb_*` physics is y-up with gravity of -9.81, while `soft_*` is y-down.
- `delta_time()` is not implemented. Subtract successive `time_now()` values.

Versions

Check which `ling` you are running with the install location and the build date of the executable, and update with `cargo install ling-lang lingfu --force`. A few builtins added recently (`str_to_hex`, `hex_to_str`, `net_local_ip`, `rand_seed`, `clipboard_set`, `clipboard_get`) are missing from older installs: they return `()` on the JIT and report `unknown function` on the interpreter.