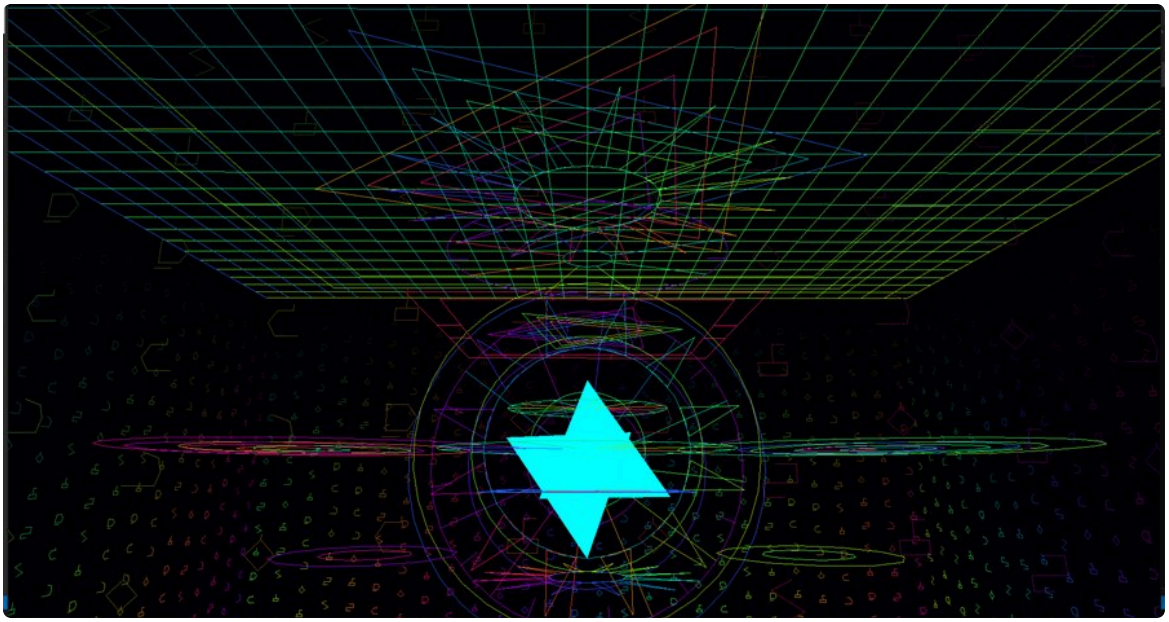
BUILD WITH LING



灵 · 靈 · 령 · āง

Build with Ling

Intermediate Edition

LING PRESS

Build with Ling: Intermediate 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 6

Preface 7

How to Use This Book 11

BUILD REAL PROGRAMS 16

1. Forms and Choices 17
2. Errors and Results 35
3. Modules and Project Structure 53
4. Data and Files 70
5. Building UI Apps 90
6. Animation and Input 108
7. Games and Physics 126
8. Music Tools 145
9. Testing and Debugging 166
10. Packaging and Shipping 189

RECIPES 204

11. Core Recipes 205
12. Graphics and UI Recipes 241
13. Audio and Game Recipes 269
14. Music Recipes: A DAW in Ling 293
15. Tooling Recipes: Editor Apps 325

REFERENCE 357

16. Keywords 358
17. Syntax, Types and Semantics 379
18. Core Builtins Reference 397
19. Command Line Tools: `ling` and `lingfu` 452
20. Window, Frame Loop, Input and Camera 467
21. Shapes, Vector Textures, Pixel Textures and Effects 494
22. Vector Fonts, the UI Toolkit and Dialogs 513
23. Animation: Anima 535
24. Audio, Music and the DAW 545
25. Physics, Game Helpers and the Body Toolkit 605
26. Input, Microphone and Speech-to-Text 641

27. The Graphics Crates 654
28. The Audio, Physics and Input Crates 672

SHOWCASE **695**

29. Case Studies: Applications Built in Ling 696
Known Limitations and Gotchas 712

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.

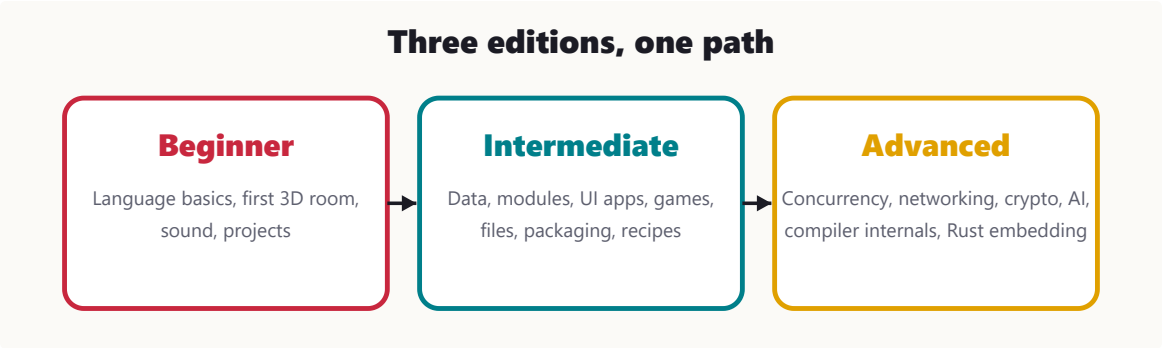


Figure: The Beginner edition covers the language basics, your first 3D room, sound and projects. Intermediate covers data, modules, UI apps, games, files, packaging and recipes. Advanced covers concurrency, networking, crypto, AI, compiler internals and Rust embedding.

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

Build Real Programs

Records, errors, modules, data, interfaces, games, music tools, testing and shipping.

CHAPTER 1

Forms and Choices

What you'll learn

Until now your data has lived in loose variables and lists of numbers. This chapter introduces the two tools Ling gives you for modelling real things: `form`, which bundles named fields into a record, and `choose`, which declares a value that is exactly one of several named variants. Together with `match` they let you write programs whose shape mirrors the problem: an inventory of items, a drawing made of circles and rectangles, a game world full of players, monsters and pickups.

You will also meet the two traps that catch almost everyone: a trailing-comma rule in `form` declarations, and the fact that `==` never says two records are equal. Both have simple fixes, and by the end you will reach for them without thinking.

Forms: records with named fields

A `form` declares a record type. You list the field names, and Ling gives you a constructor with the same name as the form. You call it like a function, and you read fields with a dot.

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

bind start = do {
  bind bolt = Item("bolt", 120, 0.25)
  println(bolt)
  println(bolt.name)
  println("stock value: ", bolt.qty * bolt.price)
}
```

```
OUTPUT
Item { name: bolt, qty: 120, price: 0.25 }
bolt
stock value: 30
```

Notice what the printout tells you: a record prints as its form name followed by every field. That is handy for debugging, and you will use `println(record)` constantly while developing.

A few rules to keep in your head:

- The constructor takes its arguments in declaration order. There is no `Item { name: "bolt" }` literal syntax; that is a parse error.
- Fields are looked up by name when you read them. Reading a field that does not exist is an error on the interpreter (`no field 'nope' on Item`) and a silent `()` on the default JIT backend.
- Records are immutable in practice. Ling has no assignment statement, so there is no `bolt.qty = 5`. To "change" a record you build a new one.

That last point shapes how you write code, so let's make it concrete.

Updating a record means building a new one

The idiom is a small function that takes the old record plus the change and returns a fresh record. Copy the fields you are keeping, replace the ones you are changing.

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

fn restock(it, n) {
  Item(it.name, it.qty + n, it.price)
}

fn sell(it, n) {
  Item(it.name, max(0, it.qty - n), it.price)
}

bind start = do {
  bind nut = Item("nut", 40, 0.1)
  println(nut)
  bind nut = restock(nut, 60)
  println(nut)
  bind nut = sell(nut, 130)
  println(nut)
}
```

```
OUTPUT
Item { name: nut, qty: 40, price: 0.1 }
Item { name: nut, qty: 100, price: 0.1 }
Item { name: nut, qty: 0, price: 0.1 }
```

`bind nut = restock(nut, 60)` looks like assignment, but it is a rebinding: the name `nut` now refers to the new record, and the old one is simply gone. Because every update goes through a function, you also get one obvious place to put rules. `sell` clamps at zero with `max`, so stock can never go negative, and no caller can forget to check.

> **TIP**
>
> Name your update functions after the business event (`restock`, `sell`, `take_damage`), not after the mechanics (`set_qty`). When the rules change, you change one function.

Lists of records

A list of records is Ling's table. You build it with a literal, read rows by index, and walk it with a `for` loop. Remember from earlier chapters that a `for` body gets a fresh scope each iteration, so a running total must use `while`.

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

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

fn total_value(items) {
  bind i = 0
  bind sum = 0
  while i < len(items) {
    bind sum = sum + line_value(items[i])
    bind i = i + 1
  }
  sum
}

bind start = do {
  bind stock = [
    Item("bolt", 120, 0.25),
    Item("nut", 200, 0.1),
    Item("washer", 80, 0.05),
    Item("bracket", 12, 3.5)
  ]
  for it in stock {
    println(it.name, ": ", line_value(it))
  }
  println("total: ", total_value(stock))
}
```

```
OUTPUT
bolt: 30
nut: 20
washer: 4
bracket: 42
total: 96
```

The last expression of a function is its result, which is why `sum` sits alone on the final line. Notice also that `for it in stock` is perfect for reading and printing. Only accumulation needs `while`.

Replacing one row

Because records and lists are values, "edit row 2" is `list_set` with an updated record. Always rebind the result.

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

fn sell(it, n) { Item(it.name, max(0, it.qty - n), it.price) }

fn find_index(items, name) {
  bind i = 0
  bind found = -1
  while i < len(items) {
    if items[i].name == name { bind found = i }
    bind i = i + 1
  }
  found
}

fn sell_by_name(items, name, n) {
  bind k = find_index(items, name)
  if k < 0 {
    items
  } else {
    list_set(items, k, sell(items[k], n))
  }
}

bind start = do {
  bind stock = [Item("bolt", 120, 0.25), Item("nut", 200, 0.1)]
  bind stock = sell_by_name(stock, "nut", 150)
  bind stock = sell_by_name(stock, "gear", 1)
  println(stock)
}
```

```
OUTPUT
[Item { name: bolt, qty: 120, price: 0.25 }, Item { name: nut, qty: 50, price: 0.1 }]
```

`find_index` returns -1 when nothing matches, the same soft-failure convention as the built-in `str_find`. Selling an unknown item leaves the list untouched. The next chapter covers more robust ways to report that kind of failure.

The trailing-comma trap

You can annotate fields with types, `form Pt { x: number, y: number }`, and Ling will accept the syntax and then ignore it. But there is a catch in the parser: a type annotation on the **last** field swallows the closing brace unless the field is followed by a comma. Here is the failure.

`trap.ling`

```
FRAGMENT
form Pt { x: number, y: number }

bind start = do {
  bind p = Pt(1, 2)
  println(p)
}
```

```
OUTPUT
$ ling run trap.ling
error[parse]: expected identifier, got LBrace
 --> trap.ling
```

The error message says nothing about commas, which is why this one is so confusing. The fix is a single character.

```
form Pt { x: number, y: number, }

bind start = do {
  bind p = Pt(1, 2)
  println(p)
}
```

```
OUTPUT
Pt { x: 1, y: 2 }
```

Plain untyped fields do not need the trailing comma, as the `Item` examples show. The safest habit is to follow the last field with a comma every time, typed or not: it is always legal, and it makes adding a field later a one-line change.

> **WARNING**
>
> Type annotations in Ling are documentation only. `x: number` does not check anything, and passing text where a number was declared will not raise an error. Use them to help the reader, and rely on tests, not annotations, to catch mistakes.

Why `==` on records is always false

This is the second trap, and it can silently ruin a game or a data program. Comparing two records, two lists or two variants with `==` never tells you that two values with identical contents are equal.

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

bind start = do {
  bind a = Item("bolt", 10, 0.25)
  bind b = Item("bolt", 10, 0.25)
  bind c = a
  println(a == b)
  println(a == a)
  println(c == a)
  println([1, 2] == [1, 2])
  println(a.name == b.name)
}
```

```
OUTPUT
false
true
false
false
true
```

Two separate records with identical contents are never equal. What happens when you compare a record with *itself* is not even consistent: the default JIT backend printed `true` for `a == a`, while `ling run --interp` prints `false`, and `c == a` (a second name for the same record) is `false` on both. Treat `==` on compound values as meaningless. Text, numbers and booleans compare by value; records, lists and variants do not compare at all. So you compare **fields**, and you wrap the comparison in a function with a good name.

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

fn same_item(a, b) {
  a.name == b.name && a.qty == b.qty && a.price == b.price
}

fn same_product(a, b) { a.name == b.name }

bind start = do {
  bind a = Item("bolt", 10, 0.25)
  bind b = Item("bolt", 12, 0.25)
  println(same_item(a, b))
  println(same_product(a, b))
  println(same_item(a, Item("bolt", 10, 0.25)))
}
```

```
OUTPUT
false
true
true
```

Writing two functions is the real lesson. "Equal" is a question about your problem: are two inventory lines the same *product*, or exactly the same *record*? Ling will not guess; you say which.

> **NOTE**
>
> The same rule applies to lists: `[1, 2] == [1, 2]` is false. To compare lists, write a loop that checks the lengths and then each element.

Formatting money and other numbers

Printing `2.5` where you wanted `2.50` looks amateur in a report. Ling has no format specifier for this, so write the helper once and reuse it. This one works in whole cents to avoid floating-point noise.

```
fn pad2(n) {
  if n < 10 { "0" + str(n) } else { str(n) }
}

fn money(x) {
  bind cents = round(x * 100)
  bind whole = floor(cents / 100)
  "$" + str(whole) + "." + pad2(cents % 100)
}

bind start = do {
  println(money(2.5))
  println(money(0.1 + 0.2))
  println(money(1234.567))
  println(money(7))
}
```

```
OUTPUT
$2.50
$0.30
$1234.57
$7.00
```

We will keep using `money` through the rest of the book. Notice that `if ... else ...` is an expression here: both branches produce a value, and the whole thing is the function's result.

Choose: values that are one of several things

A form says "a thing has all of these fields". A `choose` says "a thing is exactly one of these kinds". Each variant has a name and optionally some data.

```
choose Shape {
  Circle(r),
  Rect(w, h),
  Triangle(base, height),
  Dot
}

fn area(s) {
  match s {
    Circle(r) => 3.14159 * r * r,
    Rect(w, h) => w * h,
    Triangle(b, h) => 0.5 * b * h,
    Dot() => 0
  }
}

fn name_of(s) {
  match s {
    Circle(r) => "circle of radius " + str(r),
    Rect(w, h) => "rectangle " + str(w) + " by " + str(h),
    Triangle(b, h) => "triangle",
    Dot() => "dot"
  }
}

bind start = do {
  bind shapes = [Circle(2), Rect(3, 4), Triangle(6, 5), Dot()]
  for s in shapes {
    println(name_of(s), " -> area ", area(s))
  }
}
```

```
OUTPUT
circle of radius 2 -> area 12.56636
rectangle 3 by 4 -> area 12
triangle -> area 15
dot -> area 0
```

Three things to absorb:

1. A variant is constructed by calling its name: `Circle(2)`. A variant with no data is constructed with empty parentheses: `Dot()`.
2. `match` tries the arms top to bottom and the first that fits wins. Each arm names the variant and gives names to the data inside, which are then usable on the right-hand side of `=>`.
3. In a pattern, a variant with no data is written `Dot()`, with parentheses, to match the way you construct it. A bare `Dot` in a pattern would be an ordinary name that matches anything.

> WARNING
>
> There is no exhaustiveness check. If you add a fifth variant and forget to teach `area` about it, nothing complains: `match` falls through and the function returns `()`. Add a catch-all `_ => ...` arm when a sensible default exists, and when none does, make the fallback loud, as we do in the next chapter.

The `tag` of a variant

Every variant knows its own name, available as `.tag`. It is useful for logging, for counting how many of each kind you have, and for switching on kind when you do not need the data.

```
choose Shape { Circle(r), Rect(w, h), Dot }

bind start = do {
  bind shapes = [Circle(1), Rect(2, 2), Circle(5), Dot(), Rect(1, 9)]
  bind circles = 0
  bind i = 0
  while i < len(shapes) {
    println(i, " ", shapes[i].tag)
    if shapes[i].tag == "Circle" { bind circles = circles + 1 }
    bind i = i + 1
  }
  println("circles: ", circles)
}
```

```
OUTPUT
0 Circle
1 Rect
2 Circle
3 Dot
4 Rect
circles: 2
```

Matching on plain values

`match` is not only for variants. It works on numbers and text too, with `_` as the "anything else" arm. A bare lowercase name matches anything and binds it, which makes a handy way to name the value you are looking at.

```
fn grade(score) {
  match score {
    100 => "perfect",
    0 => "zero",
    n => if n >= 60 { "pass" } else { "fail" }
  }
}

fn command(word) {
  match word {
    "go" => 1,
    "stop" => 0,
    _ => -1
  }
}

bind start = do {
  println(grade(100), " ", grade(75), " ", grade(12), " ", grade(0))
  println(command("go"), command("stop"), command("dance"))
}
```

```
OUTPUT
perfect pass fail zero
10-1
```

Patterns are literals, names, `_` and variant patterns. There are no guards (`n if n > 60 =>`), so conditions go inside the arm with an ordinary `if`, as above.

Modelling a game world

Now combine both tools. A game world has several kinds of entity, each with different data, and a shared bag of facts about the whole world. Variants for the kinds, a form for the world.

```
choose Entity {
  Hero(name, hp),
  Monster(kind, hp, damage),
  Potion(heal),
  Wall
}

form World { turn, entities, log, }

fn hp_of(e) {
  match e {
    Hero(n, hp) => hp,
    Monster(k, hp, d) => hp,
    _ => 0
  }
}

fn hit(e, amount) {
  match e {
    Hero(n, hp) => Hero(n, max(0, hp - amount)),
    Monster(k, hp, d) => Monster(k, max(0, hp - amount), d),
    _ => e
  }
}

fn show(e) {
  match e {
    Hero(n, hp) => n + " (hero, " + str(hp) + " hp)",
    Monster(k, hp, d) => k + " (monster, " + str(hp) + " hp, hits for " + str(d) + ")",
    Potion(h) => "potion +" + str(h),
    Wall() => "wall"
  }
}

fn print_world(w) {
  println("-- turn ", w.turn)
  for e in w.entities { println("  ", show(e)) }
}
```

```
bind start = do {
  bind w = World(1, [Hero("Ada", 20), Monster("slime", 6, 3), Potion(5), Wall()], [])
  print_world(w)
  bind hero = hit(w.entities[0], 4)
  bind slime = hit(w.entities[1], 9)
  bind w = World(w.turn + 1, [hero, slime, w.entities[2], w.entities[3]], w.log)
  print_world(w)
  println("slime defeated: ", hp_of(w.entities[1]) == 0)
}
```

OUTPUT

```
-- turn 1
  Ada (hero, 20 hp)
  slime (monster, 6 hp, hits for 3)
  potion +5
  wall
-- turn 2
  Ada (hero, 16 hp)
  slime (monster, 0 hp, hits for 3)
  potion +5
  wall
slime defeated: true
```

`hit` shows the main pattern for variants with state: match on the kind, build a new value of the *same* kind with the changed field, and let everything else pass through with `_ => e`. Walls and potions cannot be hit, so they come back unchanged. The world is rebuilt by hand because records are immutable. In a real game you would write a `replace_entity(world, index, new_entity)` helper, and that is exactly what the exercises ask for.

TRY IT

Add a `Chest(gold)` variant. Run the program before you teach `show` about it. What does `show(Chest(5))` print, and why? Then add the arm.

A hand-made "maybe" type

Many languages have an option type: "either a value or nothing". Ling's soft-failure builtins return sentinels like `-1` or `""`, but your own functions can do better with a two-variant `choose`.

```
choose Found { Some(value), Nothing }

form Item { name, qty, price, }

fn lookup(items, name) {
  bind i = 0
  bind result = Nothing()
  while i < len(items) {
    if items[i].name == name { bind result = Some(items[i]) }
    bind i = i + 1
  }
  result
}

fn report(items, name) {
  match lookup(items, name) {
    Some(it) => name + ": " + str(it.qty) + " in stock",
    Nothing() => name + ": not stocked"
  }
}

bind start = do {
  bind stock = [Item("bolt", 120, 0.25), Item("nut", 0, 0.1)]
  println(report(stock, "bolt"))
  println(report(stock, "nut"))
  println(report(stock, "gear"))
}
```

```
OUTPUT
bolt: 120 in stock
nut: 0 in stock
gear: not stocked
```

The caller cannot forget the missing case, because the result is not an item; to get at the item it has to `match`. (Ling does not enforce this, but the shape of the code does.) Ling also has a built-in pair for success and failure, `ok` and `bad`, which is the subject of the next chapter.

Chapter summary

- `form Name { a, b, }` declares a record; construct with `Name(1, 2)`, read with `.a`. Records print with all their fields, which makes `println(record)` a first-class debugging tool.
- Records are never modified. Write small functions that return a fresh record, and rebind: `bind it = restock(it, 5)`.
- Put a trailing comma after the last field of a form, especially when it carries a type annotation, or the parser will fail with an unhelpful error.
- `==` on records, lists and variants is always false. Compare fields, and give each notion of equality its own function.
- `choose Name { A(x), B }` declares a value that is one of several variants. Construct with `A(1)` and `B()`; match nullary variants as `B()`.
- `match` is first-match-wins, has no guards and no exhaustiveness check, and returns `()` if nothing matches. Add `_` arms deliberately.
- `.tag` gives a variant's name as text.
- Forms for "all of these", variants for "one of these", and a form holding a list of variants gives you a world.

Exercises

1. Write `describe_stock(items)` that prints lines such as `bolt: 120 @ $0.25`, using `money` from this chapter. Which `for` or `while` form do you need if you also want a total at the end?
2. Add a `Circle(r)` perimeter function and a `scale(shape, k)` function that multiplies every dimension by `k`. What should `scale(Dot(), 3)` return?
3. Write `replace_entity(world, index, new_entity)` for the game world. It must return a new `World` with the turn unchanged. Then write `damage_all_monsters(world, amount)` using it.
4. Write `lists_equal(a, b)` for lists of numbers: same length and same element at each index. Test it on `[1,2,3]` against `[1,2,3]`, `[1,2]` and `[1,2,4]`.

Answers

1. Use a `for` loop to print each line, and a separate `while` loop (or the `total_value` function above) to accumulate. The `for` body cannot update an outer total, so mixing them in one `for` loop will print the lines but leave the total at zero. A line is `it.name + ": " + str(it.qty) + " @ " + money(it.price)`.
2. `perimeter` is `2 * 3.14159 * r` for a circle. `scale` matches each variant and rebuilds it: `Circle(r) => Circle(r * k)`, `Rect(w, h) => Rect(w * k, h * k)`, and `Dot() => Dot()`. A dot has no dimensions, so scaling it returns another dot.
3. Use `list_set`: `World(w.turn, list_set(w.entities, index, e), w.log)`. For `damage_all_monsters`, loop over the indices with `while`; when `w.entities[i].tag == "Monster"` rebind `w` with `replace_entity(w, i, hit(w.entities[i], amount))`.
4. Compare `len(a) == len(b)` first; if that fails the lists differ. Then loop with an index, recording the answer in a flag (`bind same = true`, then `bind same = false` on any mismatch), since `stop` does not break out of a loop. Return `same`.

CHAPTER 2

Errors and Results

What you'll learn

Real programs deal with bad input, missing files and impossible requests. This chapter shows how Ling expresses "this worked" and "this failed" with `ok` and `bad` results, how `match` takes them apart, and what `try` does and does not do for you. You will also learn the "soft failure" convention that Ling's own built-ins follow, and how to build assertions and a tiny logger yourself, because the language does not ship them.

Ling is candid about its limits here. There are no exceptions to catch, no `assert` and no way to exit with an error code from inside a script. What you get instead is a small set of sturdy conventions, and they are enough to write programs that fail politely.

Results: `ok` and `bad`

A result wraps either a success value or a failure reason. You create one by calling `ok(value)` or `bad(reason)`. Printing shows which one you hold.

```
fn safe_div(a, b) {
  if b == 0 {
    bad("divide by zero")
  } else {
    ok(a / b)
  }
}

bind start = do {
  println(safe_div(6, 3))
  println(safe_div(1, 0))
}
```

```
OUTPUT
Ok(2)
Err(divide by zero)
```

The type of `safe_div` is honest about its job: it does not return a number, it returns a number *or* a reason. The caller has to open the box, and `match` is how.

```
fn safe_div(a, b) {
  if b == 0 { bad("divide by zero") } else { ok(a / b) }
}

fn show(r) {
  match r {
    ok(v) => "value " + str(v),
    bad(why) => "failed: " + why
  }
}

bind start = do {
  println(show(safe_div(8, 2)))
  println(show(safe_div(8, 0)))
}
```

```
OUTPUT
value 4
failed: divide by zero
```

The patterns `ok(v)` and `bad(why)` look like variant patterns, and they work the same way: the name inside the parentheses is bound to the payload for the right-hand side of the arm. A `bad` payload is usually text, but it can be any value, including a record, which is useful when the caller needs more than a sentence (a line number, say).

Small helpers worth owning

Because `match` is the only way to look inside a result, most programs grow a handful of tiny helpers. Write them once and keep them in your library (the modules chapter shows how to share them).

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

fn unwrap_or(r, fallback) {
  match r {
    ok(v) => v,
    _ => fallback
  }
}

fn reason(r) {
  match r {
    bad(why) => why,
    _ => ""
  }
}

fn safe_sqrt(x) {
  if x < 0 { bad("sqrt of a negative number") } else { ok(sqrt(x)) }
}

bind start = do {
  println(is_ok(safe_sqrt(9)), " ", is_ok(safe_sqrt(-1)))
  println(unwrap_or(safe_sqrt(16), 0), " ", unwrap_or(safe_sqrt(-16), 0))
  println(reason(safe_sqrt(-4)))
}
```

```
OUTPUT
true false
4 0
sqrt of a negative number
```

> TIP
>
> Use `unwrap_or` only when a default is genuinely harmless, such as a missing setting falling back to a standard value. When silence would hide a bug, `match` and report.

Validating input: a worked example

`to_number` never fails loudly. Give it `"banana"` and you get `0`, which is indistinguishable from the input `"0"`. For anything user-facing you need a validator that says *why* an input is wrong. Let's build one: a function that turns text into an age between 0 and 130.

Ling has no character type, but `substr(text, start, length)` gives you one-character text, and `str_find` can test membership in a set of characters.

```
fn is_digit(c) { str_find("0123456789", c) >= 0 }

fn all_digits(text) {
  bind i = 0
  bind good = len(text) > 0
  while i < len(text) {
    if !is_digit(substr(text, i, 1)) { bind good = false }
    bind i = i + 1
  }
  good
}

fn parse_age(text) {
  bind t = trim(text)
  if len(t) == 0 {
    bad("empty input")
  } else if !all_digits(t) {
    bad("not a whole number: " + t)
  } else if to_number(t) > 130 {
    bad("too old to be true: " + t)
  } else {
    ok(to_number(t))
  }
}

fn show(r) {
  match r {
    ok(v) => "age " + str(v),
    bad(why) => "rejected (" + why + ")"
  }
}

bind start = do {
  for s in ["42", "  7 ", "", "forty", "200", "3.5"] {
    println("[", s, "] -> ", show(parse_age(s)))
  }
}
```

```
OUTPUT
[42] -> age 42
[  7 ] -> age 7
[] -> rejected (empty input)
[forty] -> rejected (not a whole number: forty)
[200] -> rejected (too old to be true: 200)
[3.5] -> rejected (not a whole number: 3.5)
```

Each rule is a separate branch, checked in order from cheapest to most specific, and each branch returns its own reason. That is the whole technique. The checks cost nothing to read, and the messages can go straight to a user.

> **NOTE**
>
> `if ... else if ... else` chains are expressions too, so the whole chain is the function's result. There is no early `return` you need to reach for.

Chaining results

When one step depends on another, you nest the matches. It is a little verbose, so factor each step into a function and let the outer function read like a recipe.

```
fn parse_num(text) {
  bind n = to_number(text)
  if n == 0 && text != "0" { bad("not a number: " + text) } else { ok(n) }
}

fn positive(n) {
  if n > 0 { ok(n) } else { bad("must be positive: " + str(n)) }
}

fn area_text(wtext, htext) {
  match parse_num(wtext) {
    bad(e) => bad("width: " + e),
    ok(w) => match parse_num(htext) {
      bad(e) => bad("height: " + e),
      ok(h) => match positive(w * h) {
        bad(e) => bad(e),
        ok(a) => ok(a)
      }
    }
  }
}

fn show(r) {
  match r {
    ok(v) => "area " + str(v),
    bad(why) => "error: " + why
  }
}

bind start = do {
  println(show(area_text("4", "2.5")))
  println(show(area_text("four", "2")))
  println(show(area_text("4", "x")))
  println(show(area_text("4", "-3")))
}
```

```
OUTPUT
area 10
error: width: not a number: four
error: height: not a number: x
error: must be positive: -12
```

Each `bad` carries a message that says exactly where the chain broke, because every layer adds a prefix such as `width:`. This is the cheap equivalent of a stack trace.

`try`: catching a runtime error

Some failures are not yours to prevent: an unknown function, a missing field, a built-in complaining about its arguments. In the interpreter these stop the program with a traceback. `try(function, args...)` runs a function and converts the outcome to a result.

```
fn plus1(x) { x + 1 }
fn broken(x) { missing_function(x) }

bind start = do {
  println(try(plus1, 4))
  println(try(broken, 4))
  match try(broken, 4) {
    ok(v) => println("fine: ", v),
    bad(e) => println("caught: ", e)
  }
}
```

```
OUTPUT
Ok(5)
Err(unknown function 'missing_function')
caught: unknown function 'missing_function'
```

Notice the argument: you pass the **function itself** (its bare name, no parentheses) and then its arguments. `try(plus1, 4)` runs `plus1(4)`.

> **WARNING**
>
> `try` only works under the interpreter. The default JIT backend cannot pass functions to built-ins, so `try` always answers `bad("try: first argument must be a function")`, even for a perfectly healthy function. The same program, run both ways:

`try_demo.ling`

```
FRAGMENT
fn plus1(x) { x + 1 }

bind start = do {
  println(try(plus1, 4))
}
```

```
OUTPUT
$ ling run try_demo.ling
Err(try: first argument must be a function)
$ ling run --interp try_demo.ling
Ok(5)
```

The same limitation applies to anything that takes a function argument (`list_map`, `list_filter`, `list_find`). So for code that must behave the same everywhere, treat `try` as a development aid and design your own functions to return `ok` and `bad` instead. That is why this chapter spends so much longer on result-returning functions than on `try`.

> **NOTE**
>
> There is a second, related catch. Under the JIT, a call to an unknown function does not fail at all: it quietly produces `()`. So a typo can sail through `ling run` and only explode under `ling run --interp`. Running your finished program once with `--interp` is the cheapest correctness check you have, and the testing chapter makes it a habit.

Soft failure: how the built-ins report problems

Most built-ins do not return results. They follow a soft-failure convention: on failure they hand back a harmless neutral value and keep going.

Built-in	On failure
`to_number("x")`	`0`
`list_get(xs, 99)`	`""` (empty text)
`str_find("abc", "z")`	`-1`
`read_file("missing.txt")`	`()` on the JIT, a runtime error on `--interp`
`file_exists(path)`	`false`

```
bind start = do {
  println(to_number("banana"))
  println(to_number("12.5"))
  println("[", list_get([1, 2, 3], 99), "]")
  println(str_find("hello", "z"))
  println(str_find("hello", "l"))
  println(file_exists("no-such-file.txt"))
  println(len(split("a,b,,c", ",")))
}
```

```
OUTPUT
0
12.5
[]
-1
2
false
4
```

The convention makes scripts forgiving, and it is also a trap: a failure and a legitimate value can look identical (`to_number("0")` and `to_number("zero")` are both `0`). Defensive coding means asking the right question *before* the call:

- Before reading a file, ask `file_exists(path)`.
- Before indexing a list, compare against `len(xs)`.
- Before trusting `to_number`, validate the text (as `parse_age` did).
- After `str_find`, compare against `-1` before using the position.

Here is a safe lookup wrapper that turns the sentinel conventions into real results, so the rest of your program can stop worrying.

```
fn get_at(xs, i) {
  if i < 0 || i >= len(xs) {
    bad("index " + str(i) + " out of range 0.." + str(len(xs) - 1))
  } else {
    ok(xs[i])
  }
}

fn find_text(haystack, needle) {
  bind k = str_find(haystack, needle)
  if k < 0 { bad("'" + needle + "' not found") } else { ok(k) }
}

fn load_text(path) {
  if file_exists(path) { ok(read_file(path)) } else { bad("no such file: " + path) }
}

fn show(r) {
  match r {
    ok(v) => "ok " + str(v),
    bad(e) => "bad " + e
  }
}

bind start = do {
  bind xs = [10, 20, 30]
  println(show(get_at(xs, 1)))
  println(show(get_at(xs, 3)))
  println(show(find_text("hello world", "wor")))
  println(show(find_text("hello world", "xyz")))
  println(show(load_text("missing.cfg")))
}
```

```
OUTPUT
ok 20
bad index 3 out of range 0..2
ok 6
bad 'xyz' not found
bad no such file: missing.cfg
```

> **TIP**
>
> Out-of-range reads behave differently by backend: `xs[99]` is quietly empty under the JIT and an error under the interpreter. Wrapping the access in `get_at` makes both behave the same, which is a good reason to use it even when you believe the index is valid.

Building assertions

Ling has no `assert`. Because functions cannot change variables outside themselves, the natural design is an assertion that **returns** a failure count (0 or 1) and prints a message when it fails. The caller adds the counts up.

```
fn check(label, condition) {
  if condition {
    println("  pass  ", label)
    0
  } else {
    println("  FAIL  ", label)
    1
  }
}

fn check_eq(label, got, want) {
  if got == want {
    println("  pass  ", label)
    0
  } else {
    println("  FAIL  ", label, ": got ", got, ", wanted ", want)
    1
  }
}

fn double(x) { x * 2 }

bind start = do {
  bind failures = 0
  bind failures = failures + check_eq("double 4", double(4), 8)
  bind failures = failures + check_eq("double 0", double(0), 0)
  bind failures = failures + check_eq("double -2", double(-2), -3)
  bind failures = failures + check("sqrt 16 is 4", sqrt(16) == 4)
  println(failures, " failure(s)")
}
```

```
OUTPUT
  pass  double 4
  pass  double 0
  FAIL  double -2: got -4, wanted -3
  pass  sqrt 16 is 4
1 failure(s)
```

The pattern scales, and it is the foundation of the test runner in the testing chapter. Two details matter. First, `check_eq` compares with `==`, so it is only for numbers, text and booleans; for records and lists compare the fields you care about. Second, floating-point sums can miss by a hair, so for decimals write a tolerance helper:

```
fn near(a, b) { abs(a - b) < 0.000001 }

bind start = do {
  println(0.1 + 0.2 == 0.3)
  println(near(0.1 + 0.2, 0.3))
}
```

```
OUTPUT
false
true
```

The first line is the surprise. The default JIT prints `false`, the usual floating-point verdict. The interpreter prints `true`, because it compares numbers with a tiny tolerance. The two backends disagree, so never rely on `==` between computed decimals; `near` gives the same answer everywhere.

A guard that stops loudly

Sometimes a bad value means the program should not continue at all. Ling has no `exit(code)` for scripts, so the practical options are to print a clear message and return early from your `start` logic, or to propagate a `bad` all the way up to a top-level `match` that prints the error. The second is easier to test and keeps `start` thin.

```
fn run_job(n) {
  match prepare(n) {
    bad(e) => bad("prepare: " + e),
    ok(v) => ok(v * 100)
  }
}

fn prepare(n) {
  if n < 0 { bad("negative job size") } else { ok(n + 1) }
}

bind start = do {
  for n in [1, -1] {
    match run_job(n) {
      ok(v) => println("job ", n, " done: ", v),
      bad(e) => println("job ", n, " aborted: ", e)
    }
  }
}
```

```
OUTPUT
job 1 done: 200
job -1 aborted: prepare: negative job size
```

A tiny logger

Logging is `println` with discipline. A level, a tag, and a place to turn the noise off are enough. Because there is no mutable global, the verbosity lives in a top-level `bind`, which every function can read.

```
bind LOG_LEVEL = 2

fn level_name(n) {
  match n {
    1 => "ERROR",
    2 => "WARN ",
    3 => "INFO ",
    _ => "DEBUG"
  }
}

fn log_at(level, tag, msg) {
  if level <= LOG_LEVEL {
    println("[", level_name(level), "] ", tag, ": ", msg)
  }
}

fn load_config(path) {
  if file_exists(path) {
    log_at(3, "config", "loading " + path)
    ok(read_file(path))
  } else {
    log_at(2, "config", "missing " + path + ", using defaults")
    bad("missing")
  }
}

bind start = do {
  log_at(1, "boot", "starting")
  log_at(3, "boot", "this INFO line is below the threshold")
  load_config("settings.cfg")
  log_at(4, "boot", "nor is this")
}
```

OUTPUT

```
[ERROR] boot: starting
[WARN ] config: missing settings.cfg, using defaults
```

Change `LOG_LEVEL` to `4` and every line appears. Because it is a plain top-level bind, a whole program can be switched between quiet and chatty with one edit. A timestamp is a one-line addition using `time_now()`, which returns seconds since the program started.

> **TRY IT**
>
> Add a fifth level `TRACE` and make `log_at` prefix each line with `round(time_now() * 1000)` milliseconds.

Chapter summary

- `ok(v)` and `bad(reason)` build results; `match` with `ok(v)` and `bad(e)` patterns takes them apart. Results print as `Ok(...)` and `Err(...)`.
- Make your own functions return results when they can fail for a reason the caller should know. Add the context at each layer with a prefix.
- `try(fn, args...)` converts a runtime error into `bad(...)`, but only on the interpreter. Under the default JIT it always returns an error, and unknown functions return `()` silently.
- Built-ins fail softly: `to_number` gives 0, `list_get` gives empty text, `str_find` gives -1. Validate first, or wrap them so they return results.
- Ling has no `assert` and no script exit code. Build `check` helpers that return failure counts, and build a logger around a top-level level constant.
- Run your finished program once with `ling run --interp` to expose errors the JIT swallows.

Exercises

1. Write `parse_pair(text)` that turns `"3,4"` into `ok([3, 4])` and rejects `"3"`, `"3,4,5"` and `"a,b"` with a different `bad` reason each. Use `split`.
2. Write `first_ok(results)` that takes a list of results and returns the first `ok`, or `bad("none succeeded")`.
3. Extend `check_eq` into `check_near(label, got, want)` that uses a tolerance, and use it to test a `circle_area(r)` function.
4. Why does `to_number("0") == to_number("zero")` hold, and how would you make a `to_number_strict` that reports the difference?

Answers

1. Split on the comma and test `len(parts)`: if it is not 2, return `bad("need exactly two values")`. Then validate each part with an `all_digits` style check, returning `bad("not numbers: " + text)`. Finally return `ok([to_number(parts[0]), to_number(parts[1])])`.
2. Walk the list with a `while` loop and a `found` result starting as `bad("none succeeded")`. Because there is no early exit, guard the update so only the first success replaces it: check `is_ok(found)` is false before rebinding.
3. `fn check_near(label, got, want) { check(label, abs(got - want) < 0.0001) }` plus a printed got/wanted pair on failure. For `circle_area` use `3.14159265 * r * r`; `pi` as a bare name is not available on the JIT, though `pi()` is.
4. Both return 0 because `to_number` has no way to say "invalid". `to_number_strict` first validates the text (allow digits, one `.`, an optional leading `-`), returns `bad("not a number: ...")` on failure, and only calls `to_number` when the text is valid.

CHAPTER 3

Modules and Project Structure

What you'll learn

A program that fits in one file is a script; a program that spans several files and shares helpers with other programs is a project. This chapter shows how to split code with `use` and `mod`, which parts of the module system work reliably and which do not, how to lay out a project with `lingfu`, and how to build a small library that two different programs share.

We will be straightforward about the rough edges. Ling's module system is deliberately small: one file pulls in another, names are global, and there are no private members. Once you accept that, a handful of naming habits make it comfortable.

Why split a program at all

Three reasons, in the order you will feel them. A 600-line file is slow to navigate. A function you wrote for one program (money formatting, a list sort) is wanted again in the next. And tests (see the testing chapter) need to load your logic without running your whole program. All three are solved by moving functions into library files and loading them.

`use`: load another file

`use "path"` loads another `.ling` file and makes everything in it available: functions, forms, variants and top-level binds. The path is in quotes, includes the extension, and is resolved **relative to the file that contains the** `use`, not to wherever you launched `ling`.

Let's build a tiny text library and use it from a program. First the library:

`lib/textkit.ling`

```
FRAGMENT
// textkit: small text-formatting helpers

fn tk_pad2(n) {
  if n < 10 { "0" + str(n) } else { str(n) }
}

fn tk_money(x) {
  bind cents = round(x * 100)
  bind whole = floor(cents / 100)
  "$" + str(whole) + "." + tk_pad2(cents % 100)
}

fn tk_pad_left(text, width) {
  bind gap = width - len(text)
  if gap > 0 { str_repeat(" ", gap) + text } else { text }
}

fn tk_pad_right(text, width) {
  bind gap = width - len(text)
  if gap > 0 { text + str_repeat(" ", gap) } else { text }
}

fn tk_rule(width) { str_repeat("-", width) }
```

Then a program that loads it:

```
use "lib/textkit.ling"

bind start = do {
  println(tk_money(1234.5))
  println(tk_pad_left("42", 6), "|")
  println(tk_pad_right("name", 8), "|")
  println(tk_rule(14))
}
```

```
OUTPUT
$1234.50
    42|
name    |
--------------
```

That is the entire mechanism. `use` is textual and global: after the line runs, `tk_money` behaves as if you had typed it into this file.

> **NOTE**
>
> The `tk_` prefix on every function is not decoration. Names from all loaded files share one namespace. If two files define a function with the same name, the one loaded **last** silently wins, and a definition in the main file beats anything it loads. There is no error and no warning. A short prefix per library keeps you out of that trouble.

You can see the collision rule directly. Two libraries define `who`, and the order of the `use` lines decides:

`lib/first.ling`

```
FRAGMENT
fn who() { "first" }
```

`lib/second.ling`

```
FRAGMENT
fn who() { "second" }
```

```
use "lib/first.ling"
use "lib/second.ling"

fn who() { "main file" }

bind start = do {
  println(who())
}
```

```
OUTPUT
main file
```

Delete the `fn who` from the main file and you get `second`; swap the `use` lines and you get `first`. Prefixes make this a non-issue.

Libraries can load libraries

A used file can itself `use` others, and each path is relative to *its own* location. So a library never needs to know where the main program lives.

`lib/listkit.ling`

FRAGMENT

```
// listkit: list helpers built from scratch (Ling has no sort or reverse)
use "textkit.ling"

fn lk_sum(xs) {
  bind i = 0
  bind total = 0
  while i < len(xs) {
    bind total = total + xs[i]
    bind i = i + 1
  }
  total
}

fn lk_reverse(xs) {
  bind out = list_new()
  bind i = len(xs) - 1
  while i >= 0 {
    bind out = list_push(out, xs[i])
    bind i = i - 1
  }
  out
}

fn lk_insert_sorted(sorted, v) {
  bind out = list_new()
  bind placed = false
  bind i = 0
  while i < len(sorted) {
    if !placed && v < sorted[i] {
      bind out = list_push(out, v)
      bind placed = true
    }
    bind out = list_push(out, sorted[i])
    bind i = i + 1
  }
  if !placed { bind out = list_push(out, v) }
  out
```

```
}

fn lk_sort(xs) {
  bind out = list_new()
  bind i = 0
  while i < len(xs) {
    bind out = lk_insert_sorted(out, xs[i])
    bind i = i + 1
  }
  out
}

fn lk_show(xs) {
  bind parts = list_new()
  bind i = 0
  while i < len(xs) {
    bind parts = list_push(parts, str(xs[i]))
    bind i = i + 1
  }
  "[" + list_join(parts, ", ") + "]"
}

fn lk_average_text(xs) {
  if len(xs) == 0 { "n/a" } else { tk_money(lk_sum(xs) / len(xs)) }
}
```

Notice `listkit` uses `tk_money` from `textkit` without caring where `textkit.ling` lives on disk, because the `use "textkit.ling"` line is relative to `lib/`.

Now two different programs that share both libraries. The first reports on a list of scores.

```ling
use "lib/listkit.ling"

bind start = do {
  bind scores = [72, 95, 58, 81, 95, 40]
  println("scores:   ", lk_show(scores))
  println("sorted:   ", lk_show(lk_sort(scores)))
  println("reversed: ", lk_show(lk_reverse(lk_sort(scores))))
  println("total:    ", lk_sum(scores))
  println("average:  ", lk_average_text(scores))
}
```

```
OUTPUT
scores:   [72, 95, 58, 81, 95, 40]
sorted:   [40, 58, 72, 81, 95, 95]
reversed: [95, 95, 81, 72, 58, 40]
total:    441
average:  $73.50
```

The second prints a price table. It loads only `textkit`, and does not care that `listkit` exists.

```ling
use "lib/textkit.ling"

form Row { item, qty, price, }

bind start = do {
  bind rows = [Row("tea", 3, 2.5), Row("biscuits", 12, 0.35), Row("kettle", 1, 24)]
  println(tk_pad_right("item", 10), tk_pad_left("qty", 4), tk_pad_left("each", 8),
tk_pad_left("total", 9))
  println(tk_rule(31))
  for r in rows {
    println(tk_pad_right(r.item, 10), tk_pad_left(str(r.qty), 4),
tk_pad_left(tk_money(r.price), 8), tk_pad_left(tk_money(r.qty * r.price), 9))
  }
}
```

```
OUTPUT
item       qty    each    total
------------------------------
tea          3   $2.50    $7.50
biscuits    12   $0.35    $4.20
kettle       1  $24.00   $24.00
```

Two programs, one set of tested helpers. When you improve `tk_money`, both benefit.

> **TIP**
>
> If your helper returns a list, always `bind` the result of `list_push`. On the default JIT backend `list_push` may modify the original list in place, while on the interpreter it returns a fresh one. Code written as `bind out = list_push(out, v)` is correct on both.
>
> **Warning:** The same rule explains why `lk_show` uses `while` rather than `for`. A `for` body gets a fresh copy of the surrounding variables on each pass, so `bind parts = list_push(...)` inside `for` is lost on the interpreter and prints `[]`, while the JIT happens to get away with it through its in-place mutation. Accumulate with `while`.

Forms and variants cross file boundaries

A `form` or `choose` declared in a library is just as shared as a function. This is how you give a project a common vocabulary.

`lib/shapes.ling`

FRAGMENT

```
choose Shape { Circle(r), Rect(w, h) }

form Labelled { label, shape, }

fn sh_area(s) {
  match s {
    Circle(r) => 3.14159 * r * r,
    Rect(w, h) => w * h,
    _ => 0
  }
}
```

```
use "lib/shapes.ling"

bind start = do {
  bind items = [Labelled("plate", Circle(10)), Labelled("tile", Rect(4, 5))]
  for it in items {
    println(it.label, ": ", sh_area(it.shape))
  }
}
```

```
OUTPUT
plate: 314.159
tile: 20
```

`mod`: a namespace inside a file

`mod name { ... }` groups functions under a name that you reach with `::`. It is the closest Ling has to a namespace and is useful when you want several `new` or `update` functions that would otherwise collide.

```
mod vec {
  fn add(a, b) { [a[0] + b[0], a[1] + b[1]] }
  fn scale(a, k) { [a[0] * k, a[1] * k] }
  fn len2(a) { vec::dot(a, a) }
  fn dot(a, b) { a[0] * b[0] + a[1] * b[1] }
}

mod color {
  fn add(a, b) { [min(255, a[0] + b[0]), min(255, a[1] + b[1]), min(255, a[2] + b[2])]
}
}

bind start = do {
  println(vec::add([1, 2], [3, 4]))
  println(vec::scale([1, 2], 3))
  println(vec::len2([3, 4]))
  println(color::add([200, 10, 10], [100, 10, 10]))
}
```

OUTPUT

```
[4, 6]
[3, 6]
25
[255, 20, 20]
```

Two rules from the experiments behind this chapter:

- **Inside a `mod`, call sibling functions with their qualified name** (`vec::dot`), as `len2` does. A bare `dot(a, a)` would not be found on the interpreter.
- A `mod` can call top-level functions of the file it lives in without any qualification.

> **WARNING**
>
> Do not put `bind` constants inside a `mod`. Reading `vec::ORIGIN` from outside gives `()` on the JIT and an "undefined" message on the interpreter. If a module needs a constant, expose it as a function: `fn origin() { [0, 0] }`, called as `vec::origin()`.

Top-level `bind` constants in a file loaded with `use` did work in the version of Ling used for this book (both backends), and the libraries above rely on that in the next section. Still, a constant exposed as a function is the choice that has never failed anywhere, so use it for anything you plan to ship.

What to share through files, and what not to

Because everything is global and nothing is private, a few conventions carry the weight a module system would normally carry:

1. **One prefix per library** (`tk_`, `lk_`, `sh_`). It is your namespace.
2. **Pure functions only in libraries.** A function that takes values and returns a value is easy to test and to reuse. A function that opens a window is a program, not a library.
3. **Constants as functions** when they must be portable (`fn tk_tab() { 4 }`).
4. **No top-level code.** A library file must not contain a `start` bind. Only the main program does; this is also what lets a test file load a library safely.
5. **Forms and variants belong next to the functions that handle them**, in the same library.

Another rule that follows from Ling's scoping: a library function sees top-level binds of the file it is defined in and loaded files, but never the caller's local variables. Anything a library needs must come in through its parameters.

A project with `lingfu`

`lingfu new <name>` creates a project directory. Here is the real output of the command, and the layout it produces.

```
OUTPUT
$ lingfu new shop
~ Ling Manifest — Spirit Token Descends ~
Creating project: shop

✓ Spirit Realm created!
  cd shop
  lingfu build
  lingfu run
```

```
OUTPUT
shop/
  Ling.toml          manifest
  README.md
  ling.ignore
  lexicons/en.ling   language data (leave alone)
  build/             output goes here
  src/start.ling     entry point
  tests/test.ling    a starter test
```

Our own layout grows from this by adding a `lib` folder under `src`:

```
OUTPUT
shop/
  Ling.toml
  src/
    start.ling
    lib/
      textkit.ling
      listkit.ling
  tests/
    test.ling
```

Create the two library files again under `shop/src/lib/` (they are the same files as above) and replace the entry point:

`shop/src/lib/textkit.ling`

FRAGMENT

```
// textkit: small text-formatting helpers

fn tk_pad2(n) {
  if n < 10 { "0" + str(n) } else { str(n) }
}

fn tk_money(x) {
  bind cents = round(x * 100)
  bind whole = floor(cents / 100)
  "$" + str(whole) + "." + tk_pad2(cents % 100)
}

fn tk_pad_left(text, width) {
  bind gap = width - len(text)
  if gap > 0 { str_repeat(" ", gap) + text } else { text }
}

fn tk_pad_right(text, width) {
  bind gap = width - len(text)
  if gap > 0 { text + str_repeat(" ", gap) } else { text }
}

fn tk_rule(width) { str_repeat("-", width) }
```

`shop/src/start.ling`

```
FRAGMENT
use "lib/textkit.ling"

form Line { item, qty, price, }

fn line_total(l) { l.qty * l.price }

bind start = do {
  bind order = [Line("tea", 3, 2.5), Line("kettle", 1, 24)]
  bind i = 0
  bind sum = 0
  while i < len(order) {
    println(tk_pad_right(order[i].item, 10),
tk_pad_left(tk_money(line_total(order[i])), 9))
    bind sum = sum + line_total(order[i])
    bind i = i + 1
  }
  println(tk_rule(19))
  println(tk_pad_right("total", 10), tk_pad_left(tk_money(sum), 9))
}
```

Now `lingfu run` finds `src/start.ling` by itself:

```
OUTPUT
$ cd shop && lingfu run
Running ...\shop\src\start.ling...
tea           $7.50
kettle       $24.00
-------------------
total        $31.50
```

`lingfu test` runs `tests/*.ling`. A test file can load the same library by walking up out of `tests`, since paths are relative to the file:

`shop/tests/test.ling`

FRAGMENT

```
use "../src/lib/textkit.ling"

fn check_eq(label, got, want) {
  if got == want { println("  pass  ", label) } else { println("  FAIL  ", label, ":
got ", got) }
}

bind test = do {
  check_eq("money rounds", tk_money(2.499), "$2.50")
  check_eq("pad left", tk_pad_left("7", 3), "  7")
  check_eq("rule", tk_rule(3), "---")
}
```

OUTPUT

```
$ cd shop && lingfu test
Testing Ling project...
  test: ...\shop\tests\test.ling
  pass  money rounds
  pass  pad left
  pass  rule
✓ All tests passed!
```

The test program binds `test` rather than `start`. Any `bind` of a `do` block counts as an entry point, which is how `lingfu test` can run every file under `tests/`. The testing chapter returns to what "failing" means for such a test.

The manifest

`Ling.toml` is created for you. For running and testing you can ignore it. It starts to matter when you package the program, which the last chapter covers; the one line worth adding early is the entry point, because `lingfu build` looks for `main.ling` unless you tell it otherwise:

```
OUTPUT
[spirit]
name = "shop"
kind = "bin"
version = "2030.0.0"
entry = "src/start.ling"
```

TRY IT

Add `lk_sum` to the shop program by loading `lib/listkit.ling` too, and make it print the number of lines and their average price. Watch what happens if `listkit.ling` and `textkit.ling` both define a helper with the same name.

Naming habits that scale

- **Functions:** `verb_noun` (`parse_age`, `draw_panel`), predicates start with `is_` or `has_`.
- **Constants: UPPER_SNAKE** when top-level, or functions when exported.
- **Prefix libraries**, not application code. The main program may use plain names, because its definitions override libraries.
- **Keep each file under about 300 lines** and each function under about 30. A function you cannot summarise in one sentence wants splitting.
- **One screen, one idea.** Put the `start` block at the *bottom* of the main file, so the file reads from helpers up to the story.

Chapter summary

- `use "relative/path.ling"` loads another file; paths are relative to the file containing the `use`, and libraries may load libraries.
- All names are global. The last definition loaded wins, and the main file beats everything. Prefix your library functions.
- Functions, forms, variants and (in the tested version) top-level binds of a loaded file are all visible. Do not rely on binds inside a `mod`.
- `mod name { fn f() {...} }` gives `name::f()`; call siblings inside a mod with the qualified name.
- A library is a file of pure functions with no `start`. A program is a file with `start`. A test is a program that loads a library.

- `lingfu new` creates a project; `lingfu run` and `lingfu test` work straight away; add `entry = "src/start.ling"` before building.

Exercises

1. Create `lib/mathkit.ling` with prefix `mk_`: `mk_clamp01`, `mk_mean(xs)` and `mk_median(xs)` (hint: sort first, using `lk_sort`). Use it from both `scores.ling` and a new program.
2. Add `tk_center(text, width)` to `textkit` and test it with odd and even padding.
3. Make two libraries collide on purpose and prove which one wins. Then rename to remove the collision.
4. Convert the constant `TAX` in a loaded file into a function and use it from a function inside a `mod`.

Answers

1. `mk_median` needs `use "listkit.ling"` at the top; sort a copy, then return the middle element, or the mean of the two middle ones when `len(xs)` is even. `mk_clamp01(x)` is `clamp(x, 0, 1)`.
2. `fn tk_center(text, width) { bind gap = width - len(text) if gap <= 0 { text } else { str_repeat(" ", floor(gap / 2)) + text + str_repeat(" ", gap - floor(gap / 2)) } }`. With odd gaps the extra space goes on the right.
3. Define the same function name in two libraries, load them in both orders, and print the result: whichever `use` line comes last wins. Rename with prefixes such as `a_who` and `b_who`.
4. `fn tax_rate() { 0.2 }`, then inside the `mod`, call `tax_rate()` (a top-level function can be called from inside a mod without qualification).

CHAPTER 4

Data and Files

What you'll learn

Programs become useful when they remember things. This chapter covers reading and writing text files, turning lines of text into records and back, building your own key-value store out of lists (Ling has no map type), and storing structured data in SQLite with the `db_*` built-ins. We finish with a complete mini project, a command-line expense tracker that persists to a CSV file.

Everything here runs headless, without a window, so you can try each example directly and compare your output with the book's.

Text files: read, write, check

Three built-ins do most of the work. `write_file(path, text)` replaces the file's contents (creating folders as needed), `read_file(path)` returns the whole file as text, and `file_exists(path)` is your safety check.

```
bind start = do {
  write_file("notes.txt", "first line\nsecond line\nthird line")
  println(file_exists("notes.txt"))
  bind text = read_file("notes.txt")
  println(len(text), " characters")
  bind lines = split(text, "\n")
  println(len(lines), " lines")
  println(lines[1])
  remove_file("notes.txt")
  println(file_exists("notes.txt"))
}
```

```
OUTPUT
true
33 characters
3 lines
second line
false
```

`split(text, separator)` turns text into a list, `list_join(list, separator)` does the reverse, and between them you can handle any line-oriented format. `remove_file` deletes, `file_size`, `list_dir`, `make_dir` and `rename_file` do what their names say.

There is no append mode. To add a line you read the old contents, add to them, and write everything back. For small files that is perfectly fine, and it keeps your data in one piece:

```
fn append_line(path, line) {
  if file_exists(path) {
    write_file(path, read_file(path) + "\n" + line)
  } else {
    write_file(path, line)
  }
}

bind start = do {
  remove_file("log.txt")
  append_line("log.txt", "boot")
  append_line("log.txt", "loaded settings")
  append_line("log.txt", "ready")
  println(read_file("log.txt"))
}
```

```
OUTPUT
boot
loaded settings
ready
```

> **WARNING**
>
> `read_file` on a missing path gives `()` on the default JIT and stops the program with a runtime error on `--interp`. Always guard with `file_exists`, or wrap the read in a function that returns `ok` / `bad` as in the errors chapter.

> **NOTE**
>
> Relative paths are resolved from the directory you ran `ling` in, not from the script's folder. If a program works from one terminal and "loses" its data from another, that is why. For a game that ships data files, build the path from a known location, or use `--pack` (see the packaging chapter).

Parsing delimited text

A CSV-like file is one record per line, fields separated by a character. We will use commas and be strict about what is allowed in a field. A full CSV parser handles quotes and escaped commas; for your own files you control the writer, so a simpler rule is more reliable: **fields may not contain the separator**. The writer replaces any comma in free text by a semicolon.

First the record, a parser that returns a result, and a writer.

```
form Expense { date, category, amount, note, }

fn clean(text) { str_replace(str_replace(text, ",", ";"), "\n", " ") }

fn to_line(e) {
  e.date + "," + e.category + "," + str(e.amount) + "," + clean(e.note)
}

fn is_number_text(t) {
  bind i = 0
  bind dots = 0
  bind ok_so_far = len(t) > 0
  while i < len(t) {
    bind c = substr(t, i, 1)
    if c == "." { bind dots = dots + 1 }
    else if str_find("0123456789", c) < 0 { bind ok_so_far = false }
    bind i = i + 1
  }
  ok_so_far && dots <= 1
}

fn from_line(line) {
  bind parts = split(line, ",")
  if len(parts) < 4 {
    bad("expected 4 fields, got " + str(len(parts)))
  } else if !is_number_text(trim(parts[2])) {
    bad("amount is not a number: " + parts[2])
  } else {
    ok(Expense(trim(parts[0]), trim(parts[1]), to_number(trim(parts[2])),
trim(parts[3])))
  }
}

bind start = do {
  bind e = Expense("2030-03-01", "food", 12.5, "lunch, with tip")
  println(to_line(e))
  println(from_line(to_line(e)))
```

```
  println(from_line("2030-03-02,food"))
  println(from_line("2030-03-02,food,lots,dinner"))
}
```

```
OUTPUT
2030-03-01,food,12.5,lunch; with tip
Ok(Expense { date: 2030-03-01, category: food, amount: 12.5, note: lunch; with tip })
Err(expected 4 fields, got 2)
Err(amount is not a number: lots)
```

The round trip works, the comma inside the note was turned into a semicolon on the way out, and malformed lines produce a `bad` that says what is wrong. Because the parser returns results, a loader can decide whether one bad line aborts the load or is skipped with a warning. We will skip.

Loading a whole file

Reading a file of records is: split into lines, skip blanks, parse each, collect the successes, and report the failures with their line numbers.

```
form Expense { date, category, amount, note, }

fn is_number_text(t) {
  bind i = 0
  bind dots = 0
  bind good = len(t) > 0
  while i < len(t) {
    bind c = substr(t, i, 1)
    if c == "." { bind dots = dots + 1 }
    else if str_find("0123456789", c) < 0 { bind good = false }
    bind i = i + 1
  }
  good && dots <= 1
}

fn from_line(line) {
  bind p = split(line, ",")
  if len(p) < 4 {
    bad("expected 4 fields")
  } else if !is_number_text(trim(p[2])) {
    bad("bad amount '" + p[2] + "'")
  } else {
    ok(Expense(trim(p[0]), trim(p[1]), to_number(trim(p[2])), trim(p[3])))
  }
}

fn keep_valid(out, r, n) {
  match r {
    ok(e) => list_push(out, e),
    bad(why) => {
      println("line ", n, " skipped: ", why)
      out
    }
  }
}

fn load_expenses(path) {
  bind out = list_new()
```

```
  if !file_exists(path) { out } else {
    bind lines = split(read_file(path), "\n")
    bind i = 0
    while i < len(lines) {
      if len(trim(lines[i])) > 0 {
        bind out = keep_valid(out, from_line(lines[i]), i + 1)
      }
      bind i = i + 1
    }
    out
  }
}

bind start = do {
  write_file("sample.csv", "2030-03-01,food,12.5,lunch\n2030-03-
01,travel,40,train\nthis line is broken\n\n2030-03-02,food,8,coffee")
  bind all = load_expenses("sample.csv")
  println(len(all), " expenses loaded")
  println(all[2])
  remove_file("sample.csv")
}
```

```
OUTPUT
line 3 skipped: expected 4 fields
3 expenses loaded
Expense { date: 2030-03-02, category: food, amount: 8, note: coffee }
```

Notice the shape of `keep_valid`. The tempting version rebinds `out` inside the arms of the `match` (`ok(e) => { bind out = list_push(out, e) }`), but a `bind` inside a `match` arm does not reach the surrounding function on the interpreter, and the default JIT only appears to get it right because `list_push` happens to mutate in place. The robust pattern is the one used here: make the `match` an expression that *returns the new list*, and bind that result outside. A brace-block arm may do work first (here, print a warning) and yield its last expression.

> **WARNING**
>
> Reserved words can bite quietly. `share` (along with `own`, `lend`, `move`, `copy`) is a keyword that Ling parses and ignores, and a variable named `share` makes the line that uses it fail to parse with the baffling message `unexpected token in expression: RParen`. If a perfectly ordinary line will not parse, check its variable names against the keyword list.

A key-value store made of lists

Ling has no dictionary type, no map and no set. When you need "look up a value by name", you build one. The simplest honest design is a list of key-value records plus a few functions. It is linear-time, which is fine up to a few thousand entries, and it is easy to save to disk.

```
form Pair { key, value, }

fn kv_new() { list_new() }

fn kv_index(store, key) {
  bind i = 0
  bind at = -1
  while i < len(store) {
    if store[i].key == key { bind at = i }
    bind i = i + 1
  }
  at
}

fn kv_has(store, key) { kv_index(store, key) >= 0 }

fn kv_get(store, key, fallback) {
  bind i = kv_index(store, key)
  if i < 0 { fallback } else { store[i].value }
}

fn kv_set(store, key, value) {
  bind i = kv_index(store, key)
  if i < 0 {
    list_push(store, Pair(key, value))
  } else {
    list_set(store, i, Pair(key, value))
  }
}

fn kv_remove(store, key) {
  bind out = list_new()
  bind i = 0
  while i < len(store) {
    if store[i].key != key { bind out = list_push(out, store[i]) }
    bind i = i + 1
  }
  out
```

```
}

bind start = do {
  bind s = kv_new()
  bind s = kv_set(s, "volume", 7)
  bind s = kv_set(s, "name", "Ada")
  bind s = kv_set(s, "volume", 9)
  println(len(s), " keys")
  println(kv_get(s, "volume", 0))
  println(kv_get(s, "colour", "none"))
  bind s = kv_remove(s, "name")
  println(kv_has(s, "name"), " ", kv_has(s, "volume"))
}
```

```
OUTPUT
2 keys
9
none
false true
```

Because every operation returns a new store, you bind the result each time: `bind s = kv_set(s, ...)`. That one habit is what makes immutable data comfortable.

> **TIP**
> For counting and totalling by category (the classic use of a map), call `kv_set(s, cat, kv_get(s, cat, 0) + amount)`. We use exactly that in the tracker.

Persisting the store

A text file of `key=value` lines is readable by humans, easy to diff and trivially recoverable. Writing it is a join; reading it is a split on the **first** `=` only, so values may themselves contain equals signs.

```
form Pair { key, value, }

fn kv_save(path, store) {
  bind lines = list_new()
  bind i = 0
  while i < len(store) {
    bind lines = list_push(lines, store[i].key + "=" + str(store[i].value))
    bind i = i + 1
  }
  write_file(path, list_join(lines, "\n"))
}

fn kv_load(path) {
  bind store = list_new()
  if file_exists(path) {
    bind lines = split(read_file(path), "\n")
    bind i = 0
    while i < len(lines) {
      bind eq = str_find(lines[i], "=")
      if eq > 0 {
        bind k = substr(lines[i], 0, eq)
        bind v = substr(lines[i], eq + 1, len(lines[i]) - eq - 1)
        bind store = list_push(store, Pair(k, v))
      }
      bind i = i + 1
    }
  }
  store
}

bind start = do {
  bind s = [Pair("volume", 7), Pair("name", "Ada"), Pair("formula", "a=b+c")]
  kv_save("prefs.cfg", s)
  println(read_file("prefs.cfg"))
  bind back = kv_load("prefs.cfg")
  println(len(back), " keys back")
  println(back[2].key, " -> ", back[2].value)
```

```
  println(back[0].value + 1)
  remove_file("prefs.cfg")
}
```

```
OUTPUT
volume=7
name=Ada
formula=a=b+c
3 keys back
formula -> a=b+c
71
```

Values come back as **text**: `"7"`, not `7`. That is why the last line prints 8 only because `+` with a number coerces; for anything non-trivial, convert on read with `to_number` after validating, as in the errors chapter.

> **NOTE**
>
> Ling does have a built-in in-process store, `shared_put(key, value)` and `shared_get(key, default)`, meant for sharing values between threads. Keys must be text. It lives only as long as the program runs, so it is not persistence, but it is handy as a global scratchpad when a function deep in your program needs a value without passing it down.

SQLite from a script

For larger or more structured data, Ling embeds SQLite. `db_open(path)` opens (or creates) a database, `db_exec(sql, params...)` runs a statement and returns the number of rows affected, and `db_query(sql, params...)` returns a list of `Row` records whose fields are the column names.

```
bind start = do {
  remove_file("demo.db")
  db_open("demo.db")
  db_exec("create table spend (id integer primary key, cat text, amt real)")
  db_exec("insert into spend (cat, amt) values (?, ?)", "food", 12.5)
  db_exec("insert into spend (cat, amt) values (?, ?)", "travel", 40)
  db_exec("insert into spend (cat, amt) values (?, ?)", "food", 8)
  bind rows = db_query("select cat, sum(amt) as total from spend group by cat order by
total desc")
  for r in rows { println(r.cat, ": ", r.total) }
  println(db_query("select count(*) as n from spend")[0].n)
  remove_file("demo.db")
}
```

```
OUTPUT
travel: 40
food: 20.5
3
```

Always pass values as `?` parameters instead of gluing them into the SQL string; it keeps quotes in user text from breaking the statement (or worse).

> **WARNING**
>
> The `db_*` functions are feature-gated. They work in the installed `ling` runner, but a natively built executable needs `ling-features = ["web"]` in its `Ling.toml`, otherwise it fails with `unknown function 'db_open'`. Also note the behaviour on errors: a bad query is a runtime error under `--interp`, but under the JIT it just returns `()`, so check `len(...)` before indexing the result.

Project: an expense tracker

We now have all the parts: a record, a parser, a writer, a store for totals and a report. The tracker keeps expenses in `expenses.csv` and takes its command from the command line, using `cli_arg(name, default)`.

`tracker.ling`

FRAGMENT

```
// tracker.ling - a tiny expense tracker
// usage: ling run tracker.ling --cmd add --date 2030-03-01 --cat food --amt 12.5 --
note lunch
//        ling run tracker.ling --cmd report

form Expense { date, category, amount, note, }
form Pair { key, value, }

bind DATA_FILE = "expenses.csv"

fn pad2(n) { if n < 10 { "0" + str(n) } else { str(n) } }
fn money(x) {
  bind c = round(x * 100)
  "$" + str(floor(c / 100)) + "." + pad2(c % 100)
}
fn pad_left(t, w) { if len(t) < w { str_repeat(" ", w - len(t)) + t } else { t } }
fn pad_right(t, w) { if len(t) < w { t + str_repeat(" ", w - len(t)) } else { t } }

fn is_number_text(t) {
  bind i = 0
  bind dots = 0
  bind good = len(t) > 0
  while i < len(t) {
    bind c = substr(t, i, 1)
    if c == "." { bind dots = dots + 1 }
    else if str_find("0123456789", c) < 0 { bind good = false }
    bind i = i + 1
  }
  good && dots <= 1
}

fn clean(t) { str_replace(str_replace(t, ",", ";"), "\n", " ") }
fn to_line(e) { e.date + "," + e.category + "," + str(e.amount) + "," + clean(e.note) }

fn from_line(line) {
  bind p = split(line, ",")
  if len(p) < 4 { bad("expected 4 fields") }
```

```
  else if !is_number_text(trim(p[2])) { bad("bad amount '" + p[2] + "'") }
  else { ok(Expense(trim(p[0]), trim(p[1]), to_number(trim(p[2])), trim(p[3]))) }
}

fn keep_valid(out, r, n) {
  match r {
    ok(e) => list_push(out, e),
    bad(why) => {
      println("warning: line ", n, " skipped (", why, ")")
      out
    }
  }
}

fn load_all() {
  bind out = list_new()
  if file_exists(DATA_FILE) {
    bind lines = split(read_file(DATA_FILE), "\n")
    bind i = 0
    while i < len(lines) {
      if len(trim(lines[i])) > 0 {
        bind out = keep_valid(out, from_line(lines[i]), i + 1)
      }
      bind i = i + 1
    }
  }
  out
}

fn save_all(items) {
  bind lines = list_new()
  bind i = 0
  while i < len(items) {
    bind lines = list_push(lines, to_line(items[i]))
    bind i = i + 1
  }
  write_file(DATA_FILE, list_join(lines, "\n"))
}
```

```
fn kv_index(s, k) {
  bind i = 0
  bind at = -1
  while i < len(s) {
    if s[i].key == k { bind at = i }
    bind i = i + 1
  }
  at
}

fn add_to(s, k, amount) {
  bind i = kv_index(s, k)
  if i < 0 { list_push(s, Pair(k, amount)) }
  else { list_set(s, i, Pair(k, s[i].value + amount)) }
}

fn cmd_add() {
  bind amt = cli_arg("amt", "")
  if !is_number_text(amt) {
    println("error: --amt must be a positive number")
  } else {
    bind items = load_all()
    bind e = Expense(cli_arg("date", "2030-01-01"), cli_arg("cat", "misc"),
to_number(amt), cli_arg("note", ""))
    bind items = list_push(items, e)
    save_all(items)
    println("added ", money(e.amount), " to ", e.category, " (", len(items), " total
entries)")
  }
}

fn cmd_report() {
  bind items = load_all()
  if len(items) == 0 {
    println("no expenses yet")
  } else {
    bind totals = list_new()
    bind grand = 0
    bind i = 0
```

```
    while i < len(items) {
      bind totals = add_to(totals, items[i].category, items[i].amount)
      bind grand = grand + items[i].amount
      bind i = i + 1
    }
    println(pad_right("category", 12), pad_left("spent", 10), pad_left("share", 8))
    println(str_repeat("-", 30))
    bind j = 0
    while j < len(totals) {
      bind pct = round(totals[j].value / grand * 100)
      println(pad_right(totals[j].key, 12), pad_left(money(totals[j].value), 10),
pad_left(str(pct) + "%", 8))
      bind j = j + 1
    }
    println(str_repeat("-", 30))
    println(pad_right("total", 12), pad_left(money(grand), 10))
  }
}

bind start = do {
  bind cmd = cli_arg("cmd", "report")
  match cmd {
    "add" => cmd_add(),
    "report" => cmd_report(),
    _ => println("unknown command '" + cmd + "' (use add or report)")
  }
}
```

It is a long listing, but every function in it appeared earlier in this chapter. Let's run a session: an empty report, a few additions (one of them wrong), and the final report.

```
OUTPUT
$ ling run tracker.ling --cmd report
no expenses yet
$ ling run tracker.ling --cmd add --date 2030-03-01 --cat food --amt 12.5 --note lunch
added $12.50 to food (1 total entries)
$ ling run tracker.ling --cmd add --date 2030-03-01 --cat travel --amt 40 --note train
added $40.00 to travel (2 total entries)
$ ling run tracker.ling --cmd add --date 2030-03-02 --cat food --amt 8 --note coffee
added $8.00 to food (3 total entries)
$ ling run tracker.ling --cmd add --cat food --amt lots
error: --amt must be a positive number
$ ling run tracker.ling --cmd report
category         spent   share
------------------------------
food            $20.50     34%
travel          $40.00     66%
------------------------------
total           $60.50
$ ling run tracker.ling --cmd dance
unknown command 'dance' (use add or report)
```

And here is the file the tracker maintains, plain text that you can open in any editor or spreadsheet:

```
OUTPUT
$ cat expenses.csv
2030-03-01,food,12.5,lunch
2030-03-01,travel,40,train
2030-03-02,food,8,coffee
```

Notice what happened to the bad amount: the command rejected it before touching the file, which is the validate-then-act rule from the errors chapter applied to a real command.

> **TRY IT**
>
> Add a `--cmd month --m 2030-03` command that reports only expenses whose date starts with the given prefix. `starts_with(text, prefix)` does the test.

Chapter summary

- `write_file`, `read_file` and `file_exists` handle text files; there is no append, so read, add and write back. Guard every read with `file_exists`.
- `split` and `list_join` convert between text and lists; `substr` and `str_find` slice around separators. Parse into records through a function that returns `ok` / `bad`.
- Choose a file format you control: no separator characters inside fields, or an escaping rule applied by the writer.
- With no map type, a list of `Pair` records plus `kv_get`, `kv_set`, `kv_remove` is a fine store; always rebind the returned store.
- Values read back from text are text. Convert on load.
- `db_open`, `db_exec` and `db_query` give you SQLite; pass values as `?` parameters; remember the build feature flag.
- `cli_arg("name", default)` makes a command-line tool out of a script.

Exercises

1. Extend `from_line` so quoted fields may contain commas: a field starting with `"` runs to the next `"`.
2. Add `--cmd delete --n 2` to the tracker, removing the second entry. What does `kv_remove`'s approach (copy everything except the match) look like for an index?
3. Build `kv_keys(store)` and `kv_sorted_keys(store)`, using the `lk_sort` idea from the modules chapter.
4. Rewrite the tracker's storage with SQLite. Which commands change? Which functions disappear?

Answers

1. Walk the characters of the line with a `while` loop and a flag `in_quotes`; split on a comma only when the flag is false, appending characters to a current-field text. Strip the surrounding quotes at the end. This is a character-level state machine; keep it in its own function that returns the list of fields.
2. Copy every row except the one at the index: `bind out = list_new()`, loop `i` over the list, push when `i != n - 1`. Validate `n` with `is_number_text` and a range check first, and report `bad`-style messages as `cmd_add` does.

3. `kv_keys` loops and pushes each `store[i].key`. For the sorted version, insert each key into a sorted list using the `lk_insert_sorted` approach; text compares with `<` on the interpreter only partially, so test it, or sort numerically by a key of your own.
4. `load_all` and `save_all` and the CSV parsing vanish. `cmd_add` becomes one `db_exec` with `?` parameters and `cmd_report` becomes one `group by` query. The rest (validation, formatting, `cli_arg`) stays.

CHAPTER 5

Building UI Apps

What you'll learn

Ling ships a toolkit of `ui_*` widgets: buttons, sliders, toggles, tabs, gauges, health bars and more. This chapter explains how an *immediate-mode* toolkit differs from the widget libraries you may know, how to organise a UI program around a state record, and how to write the small layout and label helpers that the widgets deliberately leave to you. You will build two complete programs: a settings panel with live theming, and a dashboard HUD.

Everything in this chapter opens a window, so the programs are marked `ling-gui`. Each was run for this book with a frame counter that closes the window after a dozen frames; the screenshots are captured from those runs.

Immediate mode in one page

In a retained-mode toolkit you create a button object once, attach a callback, and the library remembers it. Ling's toolkit is the opposite. Nothing is remembered between frames. **Every frame you call the widget, passing in the value it should show, and it returns the value to use next.**

```
GUI
bind start = do {
  open_window(480, 220, "Immediate mode")
  bind volume = 0.65
  bind loud = 1
  bind clicks = 0
  while window_is_open() {
    fill(8, 12, 24)

    bind volume = ui_slider(40, 50, 300, volume, 0, 1)
    bind loud = ui_toggle(40, 90, 52, 24, loud)
    if ui_button(40, 140, 120, 36) == 1 { bind clicks = clicks + 1 }

    set_color(190, 235, 255)
    ui_text(360, 44, 14, "VOL " + str(round(volume * 100)))
    ui_text(110, 94, 14, "LOUD " + str(loud))
    ui_text(176, 150, 14, "CLICKS " + str(clicks))
    present()
  }
}
```



Figure: the three widgets and their readouts. The widgets draw no text of their own, so every caption is a separate `ui_text` *call.*

Three things to take from this short program:

1. **The program owns the state.** `volume`, `loud` and `clicks` are ordinary variables. The widgets just draw them and report changes. There is nothing to synchronise and nothing to destroy.
2. **The pattern is `bind x = ui_something(..., x, ...)`.** Pass the current value in, rebind the result.
3. **Buttons return `1` on the one frame the mouse goes down over them.** Compare against `1` rather than treating the result as a boolean, and `bind clicks = clicks + 1` runs exactly once per click.

> NOTE
>
> Widgets do not draw text. A button has no label, a tab strip has no captions and a stepper shows no number. This is a design choice that keeps the toolkit independent of fonts, and it means that labels, captions and centering are your job. The rest of this chapter is mostly about doing that job well.

Colours and the whole look of a widget come from the *theme*, which we will meet later in the chapter. First, text.

Text: `ui_text`, widths and centering

`ui_text(x, y, size, text)` draws in a built-in stroke font that needs no font file. Two facts matter for layout. The `y` you pass is the **top** of the line, and glyphs fill the rows from `y` to `y + size`. The font is drawn in capitals whatever you pass in, and every glyph advances by `size * 0.86` pixels, except that the final gap does not count. So the pixel width of a string is a one-line function.

```
fn text_w(s, px) {
  len(s) * px * 0.86 - px * 0.24
}

fn centered_x(s, px, left, width) {
  left + (width - text_w(s, px)) / 2
}

bind start = do {
  println(text_w("PLAY", 16))
  println(centered_x("PLAY", 16, 40, 120))
  println(centered_x("A VERY LONG CAPTION", 16, 40, 120))
}
```

```
OUTPUT
51.2
74.4
-28.80000000000001
```

A label centred in a box therefore needs `x = centered_x(...)` and, vertically, `y = top + (height - size) / 2`. (If you use a loaded TrueType font with `font_text_fill` instead, the rule changes slightly: its `y` is also the top of the line box, but the capitals begin about 0.3 of the size lower, so a centred label sits at `y = box_top + box_height / 2 - 0.64 * size`. The stroke font is the simpler one.) The second example above shows what happens when a caption is wider than its box: the left edge goes negative. Check widths when you choose button sizes.

A button that knows its label

Now bundle the button, the caption and the centering into one helper that returns the click. This is the first of several tiny layout functions that make UI code read like a description of the interface.

```
GUI
fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

fn caption(x, y, px, s, r, g, b) {
  set_color(r, g, b)
  ui_text(x, y, px, s)
}

fn button(x, y, w, h, label) {
  bind clicked = ui_button(x, y, w, h)
  if ui_hot(x, y, w, h) {
    caption(x + (w - text_w(label, 14)) / 2, y + (h - 14) / 2, 14, label, 255, 255,
255)
  } else {
    caption(x + (w - text_w(label, 14)) / 2, y + (h - 14) / 2, 14, label, 190, 235,
255)
  }
  clicked
}

bind start = do {
  open_window(480, 200, "Buttons")
  bind last = "NOTHING YET"
  while window_is_open() {
    fill(8, 12, 24)
    if button(30, 30, 120, 40, "START") == 1 { bind last = "START" }
    if button(170, 30, 120, 40, "OPTIONS") == 1 { bind last = "OPTIONS" }
    if button(310, 30, 120, 40, "QUIT") == 1 { bind last = "QUIT" }
    caption(30, 120, 14, "LAST CLICK: " + last, 120, 255, 160)
    present()
  }
}
```

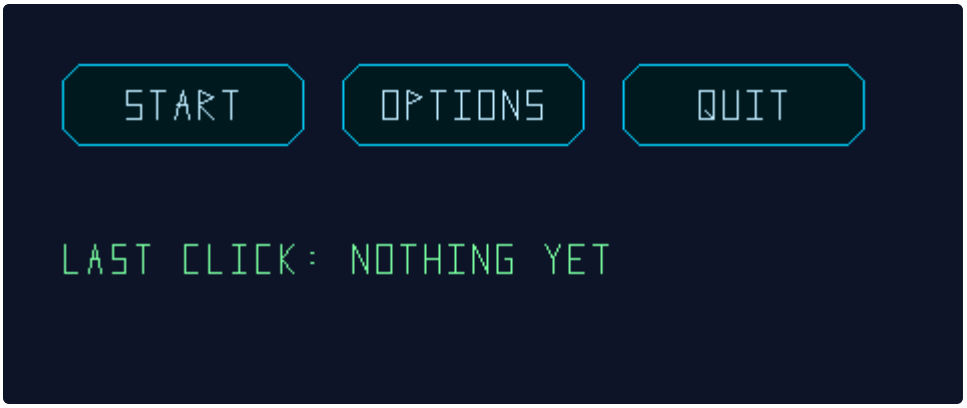


Figure: `button` returns the click and draws a centred caption, brighter while the cursor is over it.

`ui_hot(x, y, w, h)` asks whether the cursor is inside a rectangle. Here it only brightens the caption, but you can use it for tooltips and custom hover effects as well.

> **TIP**
>
> Write widget helpers with the signature `(x, y, w, h, ...)` consistently. Layout then becomes arithmetic: the next widget's `y` is `y + h + gap`. Resist hard-coding pixel positions throughout the program; compute them from a few named numbers.

Organising state: a record and an update function

With a handful of controls, loose variables are fine. With ten, you want one value that holds the whole UI state, and one function that draws the interface and returns the next state. This is the pattern for every serious Ling UI program.

```
form Settings { volume, music, sfx, quality, theme, }

fn describe(s) {
  "vol " + str(round(s.volume * 100)) + ", music " + str(s.music) + ", sfx " +
str(s.sfx) + ", quality " + str(s.quality) + ", theme " + str(s.theme)
}

fn with_volume(s, v) { Settings(v, s.music, s.sfx, s.quality, s.theme) }
fn with_quality(s, q) { Settings(s.volume, s.music, s.sfx, clamp(q, 1, 4), s.theme) }

bind start = do {
  bind s = Settings(0.5, 1, 1, 2, 0)
  println(describe(s))
  bind s = with_volume(s, 0.9)
  bind s = with_quality(s, 9)
  println(describe(s))
}
```

```
OUTPUT
vol 50, music 1, sfx 1, quality 2, theme 0
vol 90, music 1, sfx 1, quality 4, theme 0
```

The `with_` functions are the record-update idiom from the forms chapter. In the app, each frame does `bind s = settings_panel(s)`, where `settings_panel` draws the widgets and builds the new `Settings` from what they return.

Theming with `ui_theme`

Every widget draws from six theme colours: **primary**, **accent**, **track** (the dim background of meters), **warn**, **text** and **bg** (panel fill). `ui_theme` takes all six as RGB triples in that order, and `ui_theme_colors()` reads them back. Any triple you leave off keeps its previous value.

```
bind start = do {
  println(ui_theme_colors())
  ui_theme(255, 0, 128)
  println(ui_theme_colors())
}
```

```
OUTPUT
[0, 210, 255, 40, 255, 180, 44, 62, 100, 255, 90, 90, 190, 235, 255, 10, 16, 24]
[255, 0, 128, 40, 255, 180, 44, 62, 100, 255, 90, 90, 190, 235, 255, 10, 16, 24]
```

The first line is the built-in cyan theme: primary 0,210,255, accent 40,255,180 and so on. The second shows that changing only the primary left the other five alone. A theme is easiest to manage as a function that applies a whole palette:

```
FRAGMENT
fn apply_theme(i) {
  match i {
    0 => ui_theme(0, 210, 255,  40, 255, 180,  44, 62, 100,  255, 90, 90,  190, 235,
255,  10, 16, 24),
    1 => ui_theme(255, 170, 40,  255, 220, 90,  80, 56, 20,  255, 80, 60,  255, 235,
200,  24, 14, 6),
    _ => ui_theme(120, 255, 140,  200, 255, 120,  30, 70, 40,  255, 90, 90,  210, 255,
220,  6, 18, 10)
  }
}
```

Call `apply_theme` once per frame, before drawing the widgets, and the whole interface changes colour when the state changes. The widgets use the theme automatically; the text you draw yourself should read colours back from `ui_theme_colors()` if you want it to follow along.

> **WARNING**
>
> The theme is global state inside the toolkit. A helper that changes it temporarily (a red "danger" button, say) should restore the previous theme, or pass its own colour as the optional trailing `r, g, b` argument that most widgets accept. The second way is safer.

Project: a settings panel

Here is the complete panel: tabs along the top, a volume slider, a music toggle, a sound-effects checkbox, a quality stepper and an apply button. The theme tab-switches live. Study how few lines `settings_panel` needs once the helpers exist.

```
GUI
form Settings { volume, music, sfx, quality, theme, }

fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

fn apply_theme(i) {
  match i {
    0 => ui_theme(0, 210, 255,  40, 255, 180,  44, 62, 100,  255, 90, 90,  190, 235,
255,  10, 16, 24),
    1 => ui_theme(255, 170, 40,  255, 220, 90,  80, 56, 20,  255, 80, 60,  255, 235,
200,  24, 14, 6),
    _ => ui_theme(120, 255, 140,  200, 255, 120,  30, 70, 40,  255, 90, 90,  210, 255,
220,  6, 18, 10)
  }
}

fn text_color() {
  bind c = ui_theme_colors()
  set_color(c[12], c[13], c[14])
}

fn label(x, y, s) {
  text_color()
  ui_text(x, y, 14, s)
}

fn row_y(i) { 96 + i * 44 }

fn button(x, y, w, h, s) {
  bind clicked = ui_button(x, y, w, h)
  text_color()
  ui_text(x + (w - text_w(s, 14)) / 2, y + (h - 14) / 2, 14, s)
  clicked
}

fn settings_panel(s) {
  apply_theme(s.theme)
  bind c = ui_theme_colors()
```

```
  ui_panel(20, 20, 440, 330, 10)
  ui_frame(20, 20, 440, 330, 16)

  bind theme = ui_tabs(40, 40, 400, 30, 3, s.theme)
  label(52, 50, "CYAN")
  label(185, 50, "AMBER")
  label(320, 50, "GREEN")

  label(40, row_y(0), "VOLUME")
  bind volume = ui_slider(170, row_y(0) + 4, 220, s.volume, 0, 1)
  label(400, row_y(0), str(round(volume * 100)))

  label(40, row_y(1), "MUSIC")
  bind music = ui_toggle(170, row_y(1) - 4, 52, 24, s.music)

  label(40, row_y(2), "EFFECTS")
  bind sfx = ui_checkbox(170, row_y(2) - 4, 24, s.sfx)

  label(40, row_y(3), "QUALITY")
  bind quality = clamp(ui_stepper(170, row_y(3) - 6, 140, 28, s.quality, 1), 1, 4)
  label(236, row_y(3), str(quality))

  if button(150, 300, 180, 36, "RESET") == 1 {
    Settings(0.5, 1, 1, 2, theme)
  } else {
    Settings(volume, music, sfx, quality, theme)
  }
}

bind start = do {
  open_window(480, 370, "Settings")
  bind s = Settings(0.65, 1, 0, 3, 0)
  while window_is_open() {
    fill(4, 6, 12)
    bind s = settings_panel(s)
    present()
  }
}
```

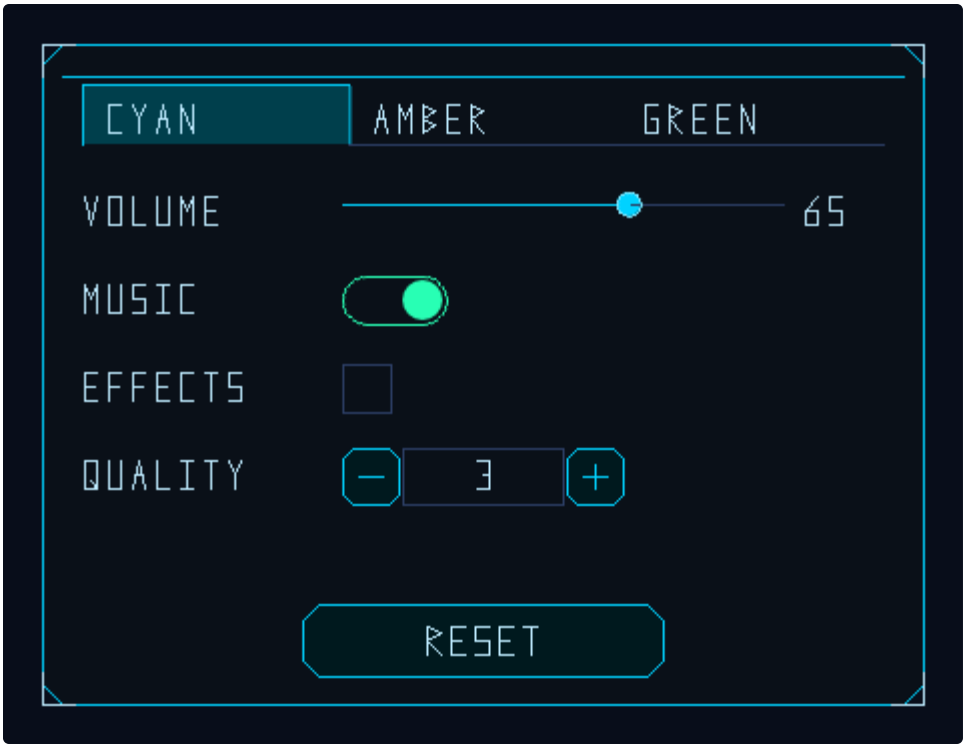


Figure: the finished settings panel in the default cyan theme. Click a tab and the whole panel recolours, because `settings_panel` *re-applies the theme every frame.*

A few lines carry the design:

- `settings_panel(s)` takes the previous state and **returns the next one**. It is the only place where drawing and state meet. `start` stays three lines long.
- `row_y(i)` is the entire layout engine: rows are 44 pixels apart starting at y = 96. Reordering the form is changing numbers, not rewriting geometry.
- `label` and `button` read the text colour from the theme, so the amber and green themes need no extra code.
- The reset button *replaces* the returned record. Because the `if` is the last expression of the function, either branch is the result.

> **TRY IT**
>
> Add a fourth tab and a fourth theme of your own, and a `ui_tooltip` that appears next to the cursor with the words `0 TO 100` while the mouse hovers the volume slider (guard it with `ui_hot`).

Project: a dashboard HUD

The second program is not about input at all. It draws a live instrument panel for an imaginary vehicle: a health bar, a ring gauge, a rolling sparkline, a compass and a radar. The widgets are all fed by `time_now()`, the seconds since the program started. This is the pattern for any game HUD: the game owns the numbers, the HUD displays them.

```
GUI
fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

fn tag(x, y, s) {
  set_color(140, 190, 215)
  ui_text(x, y, 12, s)
}

fn history(t, n) {
  bind out = list_new()
  bind i = 0
  while i < n {
    bind out = list_push(out, 50 + 30 * sin(t * 1.5 - i * 0.4) + 10 * sin(t * 4 - i))
    bind i = i + 1
  }
  out
}

bind start = do {
  open_window(640, 400, "Dashboard")
  while window_is_open() {
    bind t = time_now() + 3
    fill(5, 8, 16)

    ui_panel(16, 16, 608, 368, 12)

    tag(36, 34, "HULL")
    ui_healthbar(36, 52, 240, 22, 62 + 20 * sin(t), 100, 0)

    tag(36, 96, "FUEL")
    ui_bar(36, 114, 240, 14, 70 - fract(t / 20) * 40, 100)

    tag(36, 150, "SHIELDS")
    ui_segbar(36, 168, 240, 16, 5 + 3 * sin(t * 0.7), 10, 10)

    tag(36, 214, "SIGNAL")
    ui_spark(36, 232, 240, 60, history(t, 40))
```

```
    tag(310, 34, "REACTOR")
    ui_ring(380, 130, 52, 55 + 30 * sin(t * 0.5), 100)
    set_color(190, 235, 255)
    ui_text(380 - text_w(str(round(55 + 30 * sin(t * 0.5))), 18) / 2, 121, 18,
str(round(55 + 30 * sin(t * 0.5))))

    tag(480, 34, "HEADING")
    ui_compass(470, 62, 140, 26, fract(t / 30) * 360)

    tag(480, 100, "RADAR")
    ui_radar(545, 190, 60, t * 1.2)

    tag(310, 214, "THRUST")
    ui_battery(310, 232, 120, 26, 40 + 35 * sin(t * 0.9), 100)
    ui_counter(310, 280, 22, 34, floor(t * 37), 5)
    present()
  }
}
```

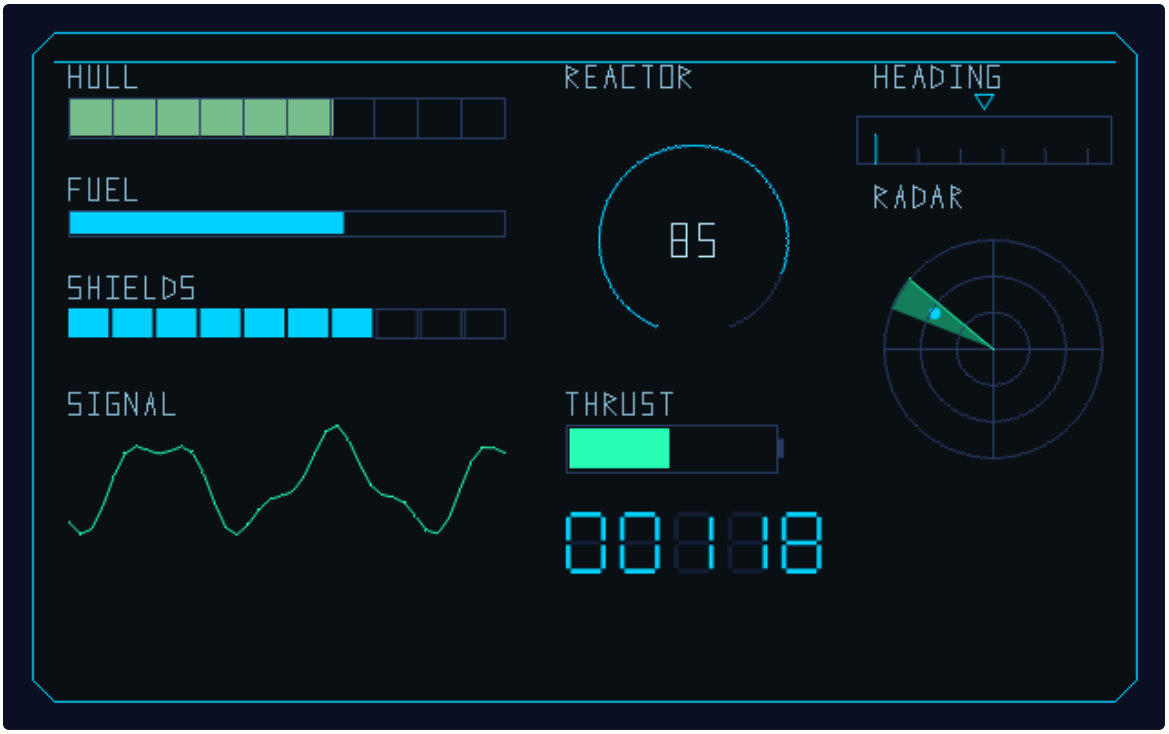


Figure: the dashboard HUD frozen at one instant. Everything on it is driven by `time_now()`.

Each widget takes a value and a maximum, and draws a proportional fill; the sparkline takes a list of values and scales them to its box. `history` builds that list on every frame, a sliding window of a wave: older samples are the same wave seen further back in time, so the line scrolls. Change `0.4` and the line stretches.

Two practical remarks about HUD code:

- **Compute once, draw once.** The ring gauge's value appears twice (the ring and the number). In a real program, `bind reactor = 55 + 30 * sin(...)` once per frame and reuse it. We duplicated the expression above only so each widget line stands alone.
- **Make a function per instrument** when the dashboard grows: `draw_fuel(x, y, level)`. Each takes a position, so the same instrument can be placed anywhere.

> NOTE
>
> Pseudo-3D variants (`ui_gauge3d`, `ui_panel3d`, `ui_radar3d`) take the same arguments plus a tilt, depth or spin, and are drawn with the same call pattern. If you can use the flat widget, you can use the 3D one.

Handling many widgets without losing your place

A UI with forty controls in a 600-line `settings_panel` is a maintenance problem. Three habits keep it tractable:

1. **One function per panel**, each taking and returning its slice of the state. `audio_panel(a)` returns an `Audio` record; the top-level panel puts the slices together.
2. **Layout through arithmetic.** `row_y(i)` for lists of rows; `col_x(i)` for columns; pass a `left` and `top` into each panel function so panels can move.
3. **One `label` helper, one `button` helper, one `section` helper.** If you notice yourself writing the same four lines of `set_color` and `ui_text` for the fifth time, it is a helper.

And one honest limitation: because the toolkit draws no text boxes, there is no text input widget. If you need a name field, you read characters yourself with `text_poll()` and draw them with `ui_text`; the input chapter shows how.

Chapter summary

- The `ui_*` toolkit is immediate mode. You call each widget every frame, pass in the current value and rebind the result: `bind v = ui_slider(x, y, w, v, lo, hi)`.
- Buttons return `1` on the click frame only; compare `== 1`.
- Widgets draw no text. Use `ui_text(x, y, size, text)`; `y` is the top of the line, width is `len * size * 0.86` less one gap, and text is capitalised.
- Centre a label with `x + (w - text_w) / 2` and `y + (h - size) / 2`. Wrap that in a `button(x, y, w, h, label)` helper that returns the click.
- Keep UI state in one record and write a panel function that takes the old state and returns the new one.
- `ui_theme` sets six colours (primary, accent, track, warn, text, bg); `ui_theme_colors()` reads them back. Re-apply the theme each frame for live switching.
- A HUD is widgets plus numbers: the game owns the numbers.

Exercises

1. Add a `ui_tooltip` to the settings panel that appears beside the cursor when it hovers the music toggle, with the caption `MUSIC ON OR OFF`.
2. Write `column_of(x, y, w, labels)` that draws one button per label stacked with an 8-pixel gap, and returns the index of the clicked one, or -1.
3. Make the dashboard's fuel bar turn red below 25 percent using the optional trailing colour arguments of `ui_bar`.
4. Add a "pause" toggle to the dashboard. When on, freeze `t` (hint: keep a separate `clock` variable and add to it only when not paused).

Answers

1. Test `ui_hot(170, row_y(1) - 4, 52, 24)` after drawing the toggle; if true, call `ui_tooltip(mouse_x() + 12, mouse_y() + 16, 170, 28)` and draw the caption with `ui_text` at the same offset plus a few pixels of padding. Draw the tooltip last so that it is on top.
2. Loop with `while`, keeping `hit = -1`; for each index call `button(x, y + i * 48, w, 40, labels[i])` and if the result is 1, `bind hit = i`. Return `hit`. Never `break` from the loop: the other buttons still need drawing, and `stop` does nothing in Ling anyway.
3. `ui_bar(x, y, w, h, value, max, r, g, b)` accepts a colour after `max`. Compute `fuel` first, then call either the red form `ui_bar(..., fuel, 100, 255, 70, 60)` or the default form from an `if` expression.

4. Declare `bind clock = 0` and `bind last = time_now()` before the loop. Each frame: `bind now = time_now()`, `if !paused { bind clock = clock + (now - last) }`, `bind last = now`. Use `clock` where the dashboard used `t`.

CHAPTER 6

Animation and Input

What you'll learn

Motion is a function of time, and interaction is a function of what the player is pressing. This chapter shows how to drive animation from the clock instead of the frame counter, how to write easing functions as ordinary Ling functions, what Anima's built-in drivers (`breathe`, `wobble`, `spring_to`, `tween_ease`) give you for free, and how to read the keyboard, mouse and gamepad. It ends with a complete moving character: acceleration, smooth following, a walking bob and an autopilot so that the program is alive the moment it starts.

The numeric experiments run headless and show their real output. The window programs are `ling-gui`; they were run with a frame limit, and the screenshots come from those runs. Input itself needs a human: those parts are parse-checked and run to completion, but I could not press keys in an automated run, and the text says so where it matters.

Time, not frames

The first rule of animation: never move something "5 pixels per frame". A faster computer then plays your game faster. Move it by

$$\textit{speed times elapsed time}$$

.

`time_now()` returns seconds since the program started, as a number with a fraction. The classic technique is to remember the previous reading and subtract.

```
bind start = do {
  bind last = time_now()
  bind i = 0
  while i < 3 {
    sleep(50)
    bind now = time_now()
    bind dt = now - last
    bind last = now
    println("frame ", i, ": slept for at least 45 ms: ", dt >= 0.045)
    bind i = i + 1
  }
}
```

```
OUTPUT
frame 0: slept for at least 45 ms: true
frame 1: slept for at least 45 ms: true
frame 2: slept for at least 45 ms: true
```

(`sleep(milliseconds)` pauses the program. We use it here only to make the clock tick measurably without a window.) The output tests `dt` instead of printing it, because the exact figure differs on every machine and every run (and may overshoot 50 ms; it never undershoots).

> **NOTE**
>
> There is no `delta_time()` in Ling; it returns nothing. `time_now()` minus the last `time_now()` is the idiom, and `frame_count()` is available if you need a frame number. With a window, `set_fps(n)` asks for a target frame rate, but your code should be correct at any rate.

With `dt` in hand, motion is one line: `x = x + speed * dt`. In Ling, spelled `bind x = x + speed * dt`.

Easing functions you can write yourself

Linear motion looks mechanical. Easing bends the progress curve: slow in, slow out, overshoot, bounce. An easing function takes progress `t` from 0 to 1 and returns an adjusted progress, also from 0 to 1 (overshoots excepted). You can write the common ones in a line

each:

```
fn clamp01(t) { clamp(t, 0, 1) }

fn ease_in(t) { t * t }
fn ease_out(t) { 1 - (1 - t) * (1 - t) }
fn ease_out_cubic(t) { 1 - pow(1 - t, 3) }
fn ease_in_out(t) {
  if t < 0.5 { 4 * t * t * t } else { 1 - pow(0 - 2 * t + 2, 3) / 2 }
}

fn ease_out_bounce(t) {
  if t < 0.363636 {
    7.5625 * t * t
  } else if t < 0.727272 {
    7.5625 * (t - 0.545454) * (t - 0.545454) + 0.75
  } else if t < 0.909090 {
    7.5625 * (t - 0.818181) * (t - 0.818181) + 0.9375
  } else {
    7.5625 * (t - 0.954545) * (t - 0.954545) + 0.984375
  }
}

fn r2(x) { round(x * 100) / 100 }

bind start = do {
  println("t      in    out   cubic in-out bounce")
  for k in [0, 1, 2, 3, 4] {
    bind t = k / 4
    println(t, "   ", r2(ease_in(t)), "  ", r2(ease_out(t)), "  ",
r2(ease_out_cubic(t)), "  ", r2(ease_in_out(t)), "  ", r2(ease_out_bounce(t)))
  }
}
```

```
OUTPUT
t      in    out   cubic in-out bounce
0   0  0  0  0  0
0.25   0.06  0.44  0.58  0.06  0.47
0.5   0.25  0.75  0.88  0.5  0.77
0.75   0.56  0.94  0.98  0.94  0.97
1   1  1  1  1  1
```

All five start at 0 and end at 1; what differs is how they get there. Notice `pow(0 - 2 * t + 2, 3)`: writing `0 - 2 * t` rather than `-2 * t` is the habit that is safe everywhere.

To use easing, remap time to progress and run it through the curve: start time, duration, then `t = clamp01((now - start) / duration)`. And `lerp(a, b, t)` (built in) blends between values.

Ling also gives you ready-made curves. `tween(a, b, t)` is a clamped linear blend, `tween_ease(a, b, t, name)` applies one of the Anima curves by name (`quad_out`, `cubic_in_out`, `elastic_out`, `bounce_out`, `back_out`, …), and `ease(name, t)` returns the raw curve value, which is handy for mixing colours. They agree with the hand-written versions:

```
fn r2(x) { round(x * 1000) / 1000 }

bind start = do {
  println(tween_ease(0, 100, 0.5, "quad_out"))
  println(tween_ease(0, 100, 0.5, "cubic_out"))
  println(r2(tween_ease(0, 100, 0.5, "bounce_out")))
  println(r2(tween_ease(0, 100, 0.75, "elastic_out")))
  println(r2(ease("cubic_out", 0.25)))
  println(tween(10, 20, 3))
}
```

```
OUTPUT
75
87.5
76.563
100.552
0.578
20
```

`quad_out` at half time is 75 percent of the way, as `ease_out(0.5)` computes by hand. `tween(10, 20, 3)` is 20, not 40, because `t` is clamped to the range 0..1.

> TIP
>
> Write your own easing when you need a curve Anima does not have (a custom bounce, a stepped curve, a curve with a pause). Use the built-ins when you just want a good-looking curve fast. Mixing both is normal.

Showing the curves

A picture makes the difference obvious. This program races six discs across the window, each using a different curve over a two-second loop. Time is wrapped with `fract` so the animation repeats.

```
GUI
fn label(y, s) {
  set_color(160, 200, 220)
  ui_text(20, y - 6, 12, s)
}

bind start = do {
  open_window(640, 360, "Easing")
  while window_is_open() {
    bind t = fract((time_now() + 1.1) / 2.4)
    bind p = clamp(t * 1.25, 0, 1)
    fill(8, 12, 24)

    bind names = ["LINEAR", "QUAD OUT", "CUBIC IN OUT", "BACK OUT", "BOUNCE OUT",
"ELASTIC OUT"]
    bind keys = ["linear", "quad_out", "cubic_in_out", "back_out", "bounce_out",
"elastic_out"]
    bind i = 0
    while i < 6 {
      bind y = 40 + i * 52
      bind x = tween_ease(150, 600, p, keys[i])
      label(y, names[i])
      set_color(40, 60, 95)
      draw_line(150, y, 600, y)
      set_color_hsl(i * 55 + 160, 80, 60)
      draw_disc(x, y, 14)
      bind i = i + 1
    }
    present()
  }
}
```



Figure: the same clock, six curves. The elastic and back curves overshoot their target; the bounce curve lands and rebounds.

Two details. First, `p = clamp(t * 1.25, 0, 1)` makes the discs arrive at 80 percent of the loop and rest for the remainder, which looks better than an instant restart. Second, the two `bind` lists are rebuilt each frame; that is fine for a six-item list, but in hot code hoist constants out of the loop.

Anima drivers

Anima's drivers are small functions from time to a number. You call them every frame with the current time and feed the result into a size, a position or an angle. Four are worth knowing now.

- `breathe(t, rate, depth)` returns a smooth in-and-out value around zero, for pulsing and idle motion.
- `wobble(t, freq, amp, phase)` returns a sine wave with an amplitude and a phase offset: sway, bob, flicker.
- `spring_to(pos, vel, target, stiffness, damping, dt)` advances a damped spring one step and returns `[new_pos, new_vel]`. It is the driver for follow-through: smooth cameras, tails, things that chase a target.

- `tween_ease` and `ease`, which we have already used.

`spring_to` is the one that rewards a closer look, because it has memory: you must keep the position *and* velocity between frames. Here it is on a fixed time step, run for one simulated second, chasing a target at 100:

```
fn r1(x) { round(x * 10) / 10 }

bind start = do {
  bind pos = 0
  bind vel = 0
  bind step = 0
  while step < 31 {
    if step % 5 == 0 {
      println("t=", r1(step / 30), "  pos=", r1(pos), "  vel=", r1(vel))
    }
    bind r = spring_to(pos, vel, 100, 120, 14, 1 / 30)
    bind pos = r[0]
    bind vel = r[1]
    bind step = step + 1
  }
}
```

```
OUTPUT
t=0  pos=0  vel=0
t=0.2  pos=80.9  vel=396
t=0.3  pos=103.3  vel=20.6
t=0.5  pos=101  vel=-16
t=0.7  pos=99.9  vel=-1.7
t=0.8  pos=100  vel=0.6
t=1  pos=100  vel=0.1
```

The value climbs to the target, overshoots a little, and settles. Stiffness controls how hard it pulls, damping how quickly the wobble dies. Lower damping gives a springier follow; higher gives a heavier, slower one. Because you feed `dt` in, the spring behaves the same at 30 or 144 frames per second.

> **WARNING**
>
> A spring chasing a target for more than a few frames needs `pos` and `vel` to be rebound every single frame. If you forget the `bind vel = r[1]`, the spring forgets how fast it was going and behaves like a stiff teleport. Both numbers are the state.

Reading the keyboard

Two built-ins cover the keyboard, and the difference is the single most important thing to learn about input:

- `key_down("left")` is true on **every frame the key is held**. Use it for movement.
- `key_pressed("space")` is true on **the one frame the key went down**. Use it for jump, fire, toggle, menu choices.

Key names are lowercase text: `"a"` to `"z"`, `"0"` to `"9"`, `"left"`, `"right"`, `"up"`, `"down"`, `"space"`, `"enter"`, `"escape"`, `"tab"`, `"lshift"`, `"f1"` to `"f12"`, and more. An unknown name is silently false, so a typo means a control that never responds, with no error. Test each binding.

A movement vector from the keys is the same few lines in every program. Write it as a function and return the two components:

```
fn axis(neg, pos) {
  bind v = 0
  if neg { bind v = v - 1 }
  if pos { bind v = v + 1 }
  v
}

bind start = do {
  println(axis(false, false))
  println(axis(true, false))
  println(axis(false, true))
  println(axis(true, true))
}
```

```
OUTPUT
0
-1
1
0
```

In a window program you call it as `bind dx = axis(key_down("left") || key_down("a"), key_down("right") || key_down("d"))`. Both key pairs are checked, so arrows and WASD both work, and pressing opposite keys cancels out.

> NOTE
>
> On `--wasm` (browser) builds, shift, ctrl and alt have no left/right variants: the names `"lshift"` and so on are not distinct from their right-hand twins. Use `"lshift"` and `"rshift"` in desktop-only code and test a web build separately.

Reading the mouse

`mouse_x()` and `mouse_y()` give the cursor position in window pixels, with the origin at the top-left and y increasing downward, just like drawing. `mouse_down()` is true while the left button is held, `mouse_down_right()` and `mouse_down_middle()` cover the others, and `mouse_scroll()` is the wheel movement (0 when idle). `mouse_dx()` and `mouse_dy()` are the change in position, but only meaningful after `capture_mouse()` has locked the cursor for a first-person camera.

For a one-shot click, remember last frame's state yourself. Because a function cannot see the caller's variables, the idiom is to compute the click in the main loop:

```
FRAGMENT
bind was_down = false
while window_is_open() {
  bind down = mouse_down()
  bind clicked = down && !was_down
  bind was_down = down
  if clicked { println("click at ", mouse_x(), ",", mouse_y()) }
  present()
}
```

(That excerpt is a fragment, not a program. The `ui_button` widget does this edge detection for you when all you need is a button.)

Reading a gamepad

The `pad_*` family supports up to four controllers. Call `pad_poll()` **once per frame, before any other pad call**; it returns the number of pads connected. After that, `pad_lx(0)` and `pad_ly(0)` read the left stick of player 0 from -1 to 1, `pad_button(0, "a")` is held, and `pad_pressed(0, "a")` is the single-frame press. Sticks have a dead zone problem on real hardware: a resting stick reads a little off zero, so ignore small values.

```
FRAGMENT
fn deadzone(v) {
  if abs(v) < 0.2 { 0 } else { v }
}

// each frame:
bind pads = pad_poll()
bind sx = deadzone(pad_lx(0))
bind sy = deadzone(pad_ly(0))   // note: down is positive, like screen y
```

The vertical stick axis uses screen convention (up is negative), which matches y in the window. Rumble is not available through `pad_rumble` on desktop; the older `gamepad_rumble` is the one that really vibrates. A program that works with no gamepad attached simply sees zeros, so you can always write the controller code and let the keyboard handle the rest. I could not attach a controller during verification, so treat the stick code as reference-checked, not play-tested.

Typed text

For a name field or a console, `text_poll()` folds the keys typed since the last call (with repeat and backspace) into an internal buffer and returns the whole buffer. `text_set("")` clears it and `text_get()` reads it without polling. Draw it with `ui_text` and you have a text box.

Project: a character that moves

Now we assemble the pieces: smooth acceleration, a spring-smoothed camera-less follow, a facing direction, a walking bob and an optional autopilot. Controls are arrow keys or WASD, the left stick of a gamepad, and a mouse click to walk to a point. It starts in autopilot so that the screenshot (and a first run) shows motion before you touch anything; press `P` to toggle autopilot, and any movement key turns it off.

```
GUI
form Hero { x, y, vx, vy, face, walk, }

fn deadzone(v) { if abs(v) < 0.2 { 0 } else { v } }

fn axis(neg, pos) {
  bind v = 0
  if neg { bind v = v - 1 }
  if pos { bind v = v + 1 }
  v
}

// Turn an input direction into a velocity that eases toward top speed.
fn accelerate(v, want, top, dt) {
  bind target = want * top
  bind rate = if want == 0 { 10 } else { 7 }
  v + (target - v) * clamp(rate * dt, 0, 1)
}

fn step_hero(h, dx, dy, dt) {
  bind len2 = sqrt(dx * dx + dy * dy)
  bind nx = if len2 > 1 { dx / len2 } else { dx }
  bind ny = if len2 > 1 { dy / len2 } else { dy }
  bind vx = accelerate(h.vx, nx, 220, dt)
  bind vy = accelerate(h.vy, ny, 220, dt)
  bind x = clamp(h.x + vx * dt, 20, 620)
  bind y = clamp(h.y + vy * dt, 60, 340)
  bind speed = sqrt(vx * vx + vy * vy)
  bind face = if abs(vx) > 8 { sign(vx) } else { h.face }
  Hero(x, y, vx, vy, face, h.walk + speed * dt * 0.06)
}

fn draw_hero(h) {
  bind bob = abs(sin(h.walk * 6)) * 6 * min(1, sqrt(h.vx * h.vx + h.vy * h.vy) / 60)
  // shadow
  set_color(0, 0, 0)
  set_alpha(0.35)
  draw_disc(h.x, h.y + 18, 16 - bob * 0.5)
```

```
  set_alpha(1)
  // body
  set_color(255, 190, 70)
  draw_disc(h.x, h.y - bob, 18)
  // eyes look the way we face
  set_color(20, 20, 40)
  draw_disc(h.x + h.face * 7, h.y - 4 - bob, 3)
  draw_disc(h.x + h.face * 1, h.y - 4 - bob, 3)
}

fn floor_grid() {
  set_color(22, 34, 56)
  bind i = 0
  while i <= 640 {
    draw_line(i, 40, i, 360)
    bind i = i + 40
  }
  bind j = 40
  while j <= 360 {
    draw_line(0, j, 640, j)
    bind j = j + 40
  }
}

bind start = do {
  open_window(640, 360, "Hero")
  bind hero = Hero(240, 220, 150, -60, 1, 2)
  bind auto = true
  bind goal_x = -1
  bind goal_y = -1
  bind last = time_now()
  while window_is_open() {
    bind now = time_now()
    bind dt = min(now - last, 0.05)
    bind last = now
    pad_poll()

    bind kx = axis(key_down("left") || key_down("a"), key_down("right") ||
key_down("d")) + deadzone(pad_lx(0))
```

```
    bind ky = axis(key_down("up") || key_down("w"), key_down("down") || key_down("s"))
+ deadzone(pad_ly(0))
    if key_pressed("p") { bind auto = !auto }
    if kx != 0 || ky != 0 { bind auto = false }
    if mouse_down() { bind goal_x = mouse_x()  bind goal_y = mouse_y()  bind auto =
false }

    bind dx = kx
    bind dy = ky
    if auto {
      bind dx = cos(now * 0.9)
      bind dy = sin(now * 1.3)
    } else if goal_x >= 0 && kx == 0 && ky == 0 {
      bind gx = goal_x - hero.x
      bind gy = goal_y - hero.y
      if sqrt(gx * gx + gy * gy) > 6 {
        bind dx = gx / 60
        bind dy = gy / 60
      }
    }

    bind hero = step_hero(hero, dx, dy, dt)

    fill(10, 14, 28)
    floor_grid()
    if goal_x >= 0 {
      set_color(80, 255, 160)
      draw_circle(goal_x, goal_y, 8 + 3 * wobble(now, 2, 1, 0))
    }
    draw_hero(hero)
    set_color(150, 190, 215)
    ui_text(14, 12, 12, "ARROWS OR WASD   CLICK TO WALK   P AUTOPILOT")
    present()
  }
}
```



Figure: the character in autopilot mode, a moment after start. Its eyes face the direction of travel, and the body bobs above its shadow as it walks.

Read the structure before the details. The state is one `Hero` record; `step_hero` takes the old record plus the player's wish (`dx`, `dy`, each -1 to 1) and the time step, and returns the new record. `draw_hero` only draws. Input is collected in one block of the main loop, then reduced to the single pair `dx, dy` that the simulation consumes. That is the shape of every Ling game: **gather input, step the state, draw the state.**

The details worth noticing:

- `accelerate` moves the velocity a fraction of the way toward the target each frame: `v + (target - v) * rate * dt`. This is exponential smoothing. It gives gentle starts and stops, and `rate` makes the character snappier or heavier. Releasing the keys uses a higher rate, so stopping is quicker than starting.
- **Normalising the direction** (`len2 > 1`) stops diagonal movement from being 41 percent faster.
- `min(now - last, 0.05)` caps `dt`. If the window is dragged or the program stalls for a second, the character does not teleport.
- **The walk cycle** is `walk`, a phase that advances with distance travelled (`speed * dt`), so the bob matches how fast the character is moving and freezes when it stands still.
- **Click to walk** feeds a direction toward the goal into the same pipeline as the keys, so there is only one movement code path to debug.

- The `if ... else` expressions on `bind` lines pick values without temporary variables.

> **TRY IT**
>
> Replace `accelerate` with the `spring_to` driver for the position: set the spring's target to the point the character is walking toward and let it smooth the whole path. Note that you then store `[pos, vel]` for each axis, not just a velocity.

Chapter summary

- Animate by *time*: `bind dt = now - last`, then `bind x = x + speed * dt`. Cap `dt` so a stall cannot teleport things. There is no `delta_time()`.
- Easing functions are small pure functions of progress; write your own or use `tween_ease(a, b, t, "curve")` and `ease("curve", t)`.
- `breathe`, `wobble` and `spring_to` supply idle motion and smooth following. `spring_to` returns `[pos, vel]` and needs both rebound each frame.
- `key_down` is held, `key_pressed` is one frame. Unknown key names are silently false.
- `mouse_x()`, `mouse_y()` and `mouse_down()` read the mouse; do your own edge detection for a one-shot click, or use `ui_button`.
- Gamepads need `pad_poll()` each frame. Apply a dead zone. Screen-convention y: up is negative.
- Structure games as gather input, step the state, draw the state, with the state in one record.

Exercises

1. Write `ease_in_out_sine(t)` using `cos` and compare it with `tween_ease(0, 1, t, "sine_in_out")` at five values of `t`.
2. Make a button that grows 10 percent while hovered, animating the size with `spring_to` so that it overshoots a little.
3. Add a dash to the hero: `key_pressed("space")` multiplies speed by 3 for a quarter of a second. Where does the dash timer live, since you cannot assign to a variable inside `step_hero`?
4. Add a second hero controlled by the arrow keys while the first uses WASD, using `pad_lx(1)` for a second gamepad.

Answers

1. `ease_in_out_sine(t)` is `0 - (cos(pi() * t) - 1) / 2` (use `pi()`, never bare `pi`). Print both at 0, 0.25, 0.5, 0.75 and 1; they should agree to rounding.
2. Keep `size` and `size_vel` as variables; each frame, `bind target = if ui_hot(x, y, w, h) { 1.1 } else { 1 }` and `bind r = spring_to(size, size_vel, target, 180, 12, dt)`. Draw the button at `w * size`, `h * size` with its centre unchanged. Low damping (12) makes it overshoot.
3. Add a `dash` field (seconds remaining) to the `Hero` form. `step_hero` returns `Hero(..., max(0, h.dash - dt))`, and the main loop starts a dash by building a copy of the record with `dash` set to 0.25 when the key is pressed. Velocity top-speed is `220 * (if h.dash > 0 { 3 } else { 1 })`.
4. Give the two heroes separate records and separate wish variables. Write `read_input(left, right, up, down, pad)` returning a two-item list `[dx, dy]` so that both heroes share the same code with different key names.

CHAPTER 7

Games and Physics

What you'll learn

This chapter builds a complete small game around Ling's rigid-body physics: a crate-smashing artillery game with three shots, a score, particle effects and sound. Along the way you will learn how the `rb_*` physics world works (and its two sharp edges: y points up, and there is no way to remove a body), how to structure a game loop as *gather input, update, draw*, how to model the game's states with `choose` and `match`, and how to use the `pfx_*` particle system and the audio effect builtins.

If you finished the animation chapter you already have the instincts; here they get a purpose.

The physics world in ten lines

Ling's `rb_*` functions expose a global rigid-body world. You add bodies, set gravity, step the clock, and read positions back. There is no world object to hold on to: the functions act on one shared world, and bodies are named by the integer index `rb_add` returns.

```
fn r2(x) { round(x * 100) / 100 }

bind start = do {
  rb_gravity(0, -9.81, 0)
  bind box = rb_add(0, 5, 0, 1)
  rb_impulse(box, 2, 0, 0)

  bind frame = 0
  while frame < 120 {
    rb_step(0.016)
    rb_floor(box, 0, 0.5, 0.6)
    if frame % 30 == 0 {
      bind p = rb_pos(box)
      println("t=", r2(frame * 0.016), "  x=", r2(p[0]), "  y=", r2(p[1]))
    }
    bind frame = frame + 1
  }
}
```

```
OUTPUT
t=0  x=0.03  y=5
t=0.48  x=0.99  y=3.75
t=0.96  x=1.95  y=0.57
t=1.44  x=2.73  y=1.59
```

Walk through the calls:

- `rb_gravity(gx, gy, gz)` sets the world's gravity vector. **y points up in the physics world**, so gravity is negative: `-9.81`. The engine default is already -9.81, but if you call `rb_gravity` and leave out `gy`, it defaults to `+9.81`, which makes things fall *up*. Always write all three numbers.
- `rb_add(x, y, z, mass)` creates a one-metre cube at that position and returns its index. (A resting body on the floor reports `y = 0.5`: its centre is half a metre up.)
- `rb_impulse(body, ix, iy, iz)` adds an instant change of velocity (times mass). Here the box gets 2 m/s sideways.
- `rb_step(dt)` advances every body by `dt` seconds and resolves body-to-body collisions.

- `rb_floor(body, floor_y, restitution, friction)` is a ground plane **for that one body**, and you must call it every frame, for every body that should land. Restitution is the bounciness (0 dead, 1 perfectly elastic), friction turns sideways sliding into roll.
- `rb_pos(body)` returns `[x, y, z]`; `rb_rot(body)` returns a rotation quaternion `[x, y, z, w]`.

> **WARNING**
>
> The sign of y is the classic trap. The drawing world (windows, 2D shapes, the 3D camera) has +y *down*; the physics world has +y *up*. Gravity must be negative, floors are at low y, and when you draw a body you flip: `screen_y = ground_screen_y - physics_y * scale`. A body that "falls through the floor" almost always means a gravity sign or floor-direction mistake.

What rigid bodies do not do

Two limits shape the game below, so learn them now:

- **A body cannot be removed or teleported.** There is no `rb_remove` and no `rb_set_pos`. Once added, a body exists for the life of the program. Our game adds a fresh projectile per shot and keeps a list of every body so that old ones keep landing. To "restart" you run the program again.
- **There is no velocity getter.** You can read position and rotation, so if you want speed you subtract successive positions yourself.

Bodies are boxes, so stacking works well; the next program builds a pyramid of six crates and lets it settle for two seconds, headless, before we print where everything ended up.

```
fn r2(x) { round(x * 100) / 100 }

fn tower_x(i) {
  match i {
    0 => 9, 1 => 10, 2 => 11,
    3 => 9.5, 4 => 10.5,
    _ => 10
  }
}

fn tower_y(i) {
  if i < 3 { 0.5 } else if i < 5 { 1.5 } else { 2.5 }
}

bind start = do {
  rb_gravity(0, -9.81, 0)
  bind ids = list_new()
  bind k = 0
  while k < 6 {
    bind ids = list_push(ids, rb_add(tower_x(k), tower_y(k), 0, 1))
    bind k = k + 1
  }
  bind f = 0
  while f < 125 {
    rb_step(0.016)
    bind k = 0
    while k < 6 {
      rb_floor(ids[k], 0, 0.1, 0.8)
      bind k = k + 1
    }
    bind f = f + 1
  }
  bind k = 0
  while k < 6 {
    bind p = rb_pos(ids[k])
    println("crate ", k, ": x=", r2(p[0]), " y=", r2(p[1]))
```

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

```
OUTPUT
crate 0: x=9 y=0.5
crate 1: x=10 y=0.5
crate 2: x=11 y=0.5
crate 3: x=9.5 y=1.49
crate 4: x=10.5 y=1.49
crate 5: x=10 y=2.49
```

The pyramid stays where it was built: the bottom row at y = 0.5, the next at about 1.5 and the top near 2.5. A stable stack is the precondition for a satisfying smash.

Modelling the game with choose

An arcade game is a state machine. At any moment it is in exactly one *phase*, each phase has its own data and rules, and certain events move it to the next. That is what `choose` is for. Our game has four phases:

- **Aiming**: the player sets angle and power; Space fires.
- **Flying(t, hit)**: the projectile is in the air; `t` counts seconds since launch and `hit` remembers whether the first impact effect has played.
- **Result(t)**: a short pause while the score updates; after it, back to Aiming or on to the end.
- **Over**: the final screen.

The whole game state is one record, `Game`, holding the phase, shots used, aim settings, the list of tower crates and the list of *all* bodies.

```
choose Phase { Aiming, Flying(t, hit), Result(t), Over }

form Game { phase, shots, angle, power, }

fn phase_name(g) {
  match g.phase {
    Aiming() => "aiming",
    Flying(t, hit) => "flying for " + str(t) + "s",
    Result(t) => "showing result",
    Over() => "game over"
  }
}

fn next_phase(g) {
  match g.phase {
    Aiming() => Flying(0, false),
    Flying(t, hit) => if t > 3 { Result(0) } else { Flying(t + 1, hit) },
    Result(t) => if g.shots >= 3 { Over() } else { Aiming() },
    _ => Over()
  }
}

bind start = do {
  bind g = Game(Aiming(), 0, 45, 12)
  bind i = 0
  while i < 8 {
    println(i, ": ", phase_name(g))
    bind g = Game(next_phase(g), g.shots + (if g.phase.tag == "Aiming" { 1 } else { 0
}), g.angle, g.power)
    bind i = i + 1
  }
}
```

```
OUTPUT
0: aiming
1: flying for 0s
2: flying for 1s
3: flying for 2s
4: flying for 3s
5: flying for 4s
6: showing result
7: aiming
```

That toy loop is the whole idea in miniature: a function that takes the current phase and returns the next, and one place that prints what is happening. `phase_name` and `next_phase` use `match` exactly as the forms-and-choices chapter taught, and a `.tag` check reads the phase's kind without unpacking it.

> TIP
>
> Put the phase *rules* in an `update` function (what happens each frame in each phase) and the phase *picture* in a `draw` function (how each phase looks). Neither calls the other. The main loop calls both in order. When something is wrong, you always know which half to read.

The game loop

Every game here has the same skeleton:

```
OUTPUT
setup
while the window is open:
    1. gather input        (keys, mouse, pad) into plain values
    2. update(state, input, dt) -> new state
    3. draw(state)
    present()
```

Input is gathered once and handed to `update` as ordinary numbers; `update` never calls `key_down`. That one rule makes a game testable: you can call `update` with made-up input and check the state it returns, with no window.

Particles

The `pfx_*` family is a fire-and-forget particle system with its own pool: you emit particles, step them, and draw them. `pfx_burst(x, y, z, count, speed, life, size0, size1, col0, col1, kind, gravity, vx, vy, vz)` throws `count` particles in random directions from a point. Colours are packed integers, `red * 65536 + green * 256 + blue` (Ling has no hex literals, so we write a helper). `kind` 0 is an additive glow dot, 1 a streak. `pfx_step(dt, gravity, drag, floor_y, bounce)` advances them, `pfx_draw()` queues them for drawing, and `pfx_count()` says how many are alive.

You can watch the pool without a window:

```
fn rgb(r, g, b) { r * 65536 + g * 256 + b }

bind start = do {
  pfx_clear()
  pfx_burst(0, 0, 0, 30, 5, 1, 0.2, 0, rgb(255, 200, 60), rgb(255, 60, 0), 0, 1, 0, 0,
0)
  println("after burst: ", pfx_count())
  pfx_step(0.5, 9.8, 0, 1000, 0.4)
  println("after 0.5s: ", pfx_count())
  pfx_step(1.0, 9.8, 0, 1000, 0.4)
  println("after 1.5s: ", pfx_count())
}
```

```
OUTPUT
after burst: 30
after 0.5s: 30
after 1.5s: 0
```

A burst with `life` 1 second lives between 0.6 and 1.0 s per particle, so all 30 are alive at half a second and all gone after one and a half. The pool is deterministic: it has its own fixed random seed, so the same calls make the same burst every run.

Particles live in the **3D** world, drawn with the same camera as shapes. To use them in a flat 2D game you convert from pixels. With the default camera in a 640 by 360 window, a point at depth `z` lands at `screen = centre + 360 * world / (z + 5)`. Choose `z = 4`, and one world unit is exactly 360 / 9 = 40 pixels, so a pixel position `(sx, sy)` becomes world `((sx - 320) /`

`40, (sy - 180) / 40, 4)`. That is the whole trick, written below as two one-line functions. Particles also fall with `pfx_step`'s gravity, which acts along +y, down the screen, matching the 2D drawing.

Sound effects

Ling can synthesise effects without sound files. `audio_blip(freq, duration, wave, amp)` is a short interface beep, and `audio_sfx(x, y, z, w, freq, amp, duration, wave)` plays a one-shot at a position in space. The `wave` is text: `"sine"`, `"square"`, and others. For a launch sound, a falling-pitch blip is enough; for an impact, a low square burst. Effects are fire-and-forget; no handle to keep. Since sounds cannot be shown on a page, I confirmed here only that the calls run in a windowed program without error; how they sound on your speakers is for your ears.

Project: Crate Smash

The finished game: three shots to knock down a pyramid of six crates. Left and right set power, up and down set the launch angle, Space fires. The dotted arc previews the shot without air resistance. Score is 100 per crate knocked clear of its starting place, and a perfect clear ends the game early.

Read it top to bottom: helpers, then `update`, then `draw`, then the loop.

```
GUI
choose Phase { Aiming, Flying(t, hit), Result(t), Over }

form Game { phase, shots, angle, power, tower, bodies, proj, }

// ---------- helpers ----------
fn rgb(r, g, b) { r * 65536 + g * 256 + b }

fn tower_x(i) {
  match i {
    0 => 9, 1 => 10, 2 => 11,
    3 => 9.5, 4 => 10.5,
    _ => 10
  }
}
fn tower_y(i) { if i < 3 { 0.5 } else if i < 5 { 1.5 } else { 2.5 } }

// physics (y up, metres) <-> screen (y down, pixels, 40 px per metre)
fn sx(x) { 60 + x * 40 }
fn sy(y) { 320 - y * 40 }
// screen pixels -> 3D particle space (camera depth z = 4 gives 40 px per unit)
fn px3(x) { (sx(x) - 320) / 40 }
fn py3(y) { (sy(y) - 180) / 40 }

fn build_tower() {
  bind ids = list_new()
  bind k = 0
  while k < 6 {
    bind ids = list_push(ids, rb_add(tower_x(k), tower_y(k), 0, 1))
    bind k = k + 1
  }
  ids
}

fn knocked(tower) {
  bind n = 0
  bind k = 0
  while k < len(tower) {
```

```
    bind p = rb_pos(tower[k])
    bind dx = p[0] - tower_x(k)
    bind dy = p[1] - tower_y(k)
    if sqrt(dx * dx + dy * dy) > 0.6 { bind n = n + 1 }
    bind k = k + 1
  }
  n
}

fn near_tower(tower, b) {
  bind p = rb_pos(b)
  bind hit = false
  bind k = 0
  while k < len(tower) {
    bind q = rb_pos(tower[k])
    if abs(p[0] - q[0]) < 1.1 && abs(p[1] - q[1]) < 1.1 { bind hit = true }
    bind k = k + 1
  }
  hit
}

fn floor_all(bodies) {
  bind k = 0
  while k < len(bodies) {
    rb_floor(bodies[k], 0, 0.15, 0.8)
    bind k = k + 1
  }
}

// ---------- update ----------
// turn: -1 / 0 / 1 changes the angle, push changes the power, fire launches
fn update(g, dt, turn, push, fire) {
  match g.phase {
    Aiming() => {
      bind angle = clamp(g.angle + turn * 40 * dt, 10, 75)
      bind power = clamp(g.power + push * 6 * dt, 6, 16)
      if fire {
        bind b = rb_add(1.5, 0.5, 0, 1)
        rb_impulse(b, power * cos(angle * 3.14159265 / 180), power * sin(angle *
```

```
3.14159265 / 180), 0)
        audio_blip(520, 0.12, "square", 0.15)
        Game(Flying(0, false), g.shots + 1, angle, power, g.tower, list_push(g.bodies,
b), b)
      } else {
        Game(g.phase, g.shots, angle, power, g.tower, g.bodies, g.proj)
      }
    }
    Flying(t, hit) => {
      rb_step(dt)
      floor_all(g.bodies)
      bind touched = hit || near_tower(g.tower, g.proj)
      if touched && !hit {
        bind p = rb_pos(g.proj)
        pfx_burst(px3(p[0]), py3(p[1]), 4, 40, 6, 1.2, 0.35, 0, rgb(255, 210, 90),
rgb(255, 60, 20), 0, 1, 0, 0, 0)
        audio_sfx(p[0], p[1], 0, 1, 110, 0.4, 0.25, "square")
      }
      if t > 3.2 {
        Game(Result(0), g.shots, g.angle, g.power, g.tower, g.bodies, g.proj)
      } else {
        Game(Flying(t + dt, touched), g.shots, g.angle, g.power, g.tower, g.bodies,
g.proj)
      }
    }
    Result(t) => {
      rb_step(dt)
      floor_all(g.bodies)
      if t > 1.5 {
        if g.shots >= 3 || knocked(g.tower) == 6 {
          Game(Over(), g.shots, g.angle, g.power, g.tower, g.bodies, g.proj)
        } else {
          Game(Aiming(), g.shots, g.angle, g.power, g.tower, g.bodies, g.proj)
        }
      } else {
        Game(Result(t + dt), g.shots, g.angle, g.power, g.tower, g.bodies, g.proj)
      }
    }
    _ => g
```

```
  }
}

// ---------- draw ----------
fn draw_crate(b) {
  bind p = rb_pos(b)
  bind q = rb_rot(b)
  bind a = 2 * atan2(q[2], q[3])
  bind c = cos(a)
  bind s = sin(a)
  bind pts = list_new()
  for k in [[-0.5, -0.5], [0.5, -0.5], [0.5, 0.5], [-0.5, 0.5]] {
    bind pts = list_push(pts, sx(p[0] + k[0] * c - k[1] * s))
    bind pts = list_push(pts, sy(p[1] + k[0] * s + k[1] * c))
  }
  draw_poly(pts)
}

fn draw_world(g) {
  fill(14, 20, 38)
  set_color(34, 52, 40)
  grad_rect(0, 320, 640, 40, 34, 52, 40, 34, 52, 40, 1)
  set_color(80, 120, 90)
  draw_line(0, 320, 640, 320)
  // tower crates, then projectiles
  bind k = 0
  while k < len(g.bodies) {
    if k < 6 { set_color(200, 140, 70) } else { set_color(120, 200, 255) }
    draw_crate(g.bodies[k])
    bind k = k + 1
  }
}

fn draw_aim(g) {
  bind a = g.angle * 3.14159265 / 180
  bind vx = g.power * cos(a)
  bind vy = g.power * sin(a)
  set_color(255, 255, 255)
  bind i = 1
```

```
  while i < 14 {
    bind t = i * 0.12
    draw_disc(sx(1.5 + vx * t), sy(0.5 + vy * t - 4.905 * t * t), 2)
    bind i = i + 1
  }
}

fn draw_hud(g) {
  set_color(190, 235, 255)
  ui_text(16, 14, 14, "SHOT " + str(min(3, g.shots + (if g.phase.tag == "Aiming" { 1 }
else { 0 }))) + " OF 3")
  ui_text(16, 36, 14, "SCORE " + str(knocked(g.tower) * 100))
  match g.phase {
    Aiming() => {
      ui_text(380, 14, 12, "ANGLE " + str(round(g.angle)) + "   POWER " +
str(round(g.power)))
      ui_text(380, 34, 12, "ARROWS AIM   SPACE FIRE")
    }
    Over() => {
      set_color(255, 220, 120)
      ui_text(200, 140, 28, if knocked(g.tower) == 6 { "CLEARED" } else { "GAME OVER"
})
    }
    _ => ui_text(380, 14, 12, "...")
  }
}

bind start = do {
  open_window(640, 360, "Crate Smash")
  rb_gravity(0, -9.81, 0)
  bind tower = build_tower()
  bind g = Game(Aiming(), 0, 30, 13, tower, tower, -1)
  bind demo = cli_arg("demo", "0") == "1"
  bind tick = 0
  bind last = time_now()
  while window_is_open() {
    bind now = time_now()
    bind dt = 0.016
    bind last = now
```

```
    bind tick = tick + 1

    // 1. gather input
    bind turn = (if key_down("up") { 1 } else { 0 }) - (if key_down("down") { 1 } else
{ 0 })
    bind push = (if key_down("right") { 1 } else { 0 }) - (if key_down("left") { 1 }
else { 0 })
    bind fire = key_pressed("space") || (demo && tick == 3)

    // 2. update, 3. draw
    bind g = update(g, dt, turn, push, fire)
    pfx_step(dt, 9.8, 0.3, 1000, 0.4)
    draw_world(g)
    if g.phase.tag == "Aiming" { draw_aim(g) }
    pfx_draw()
    flush_3d()
    draw_hud(g)
    present()
  }
}
```

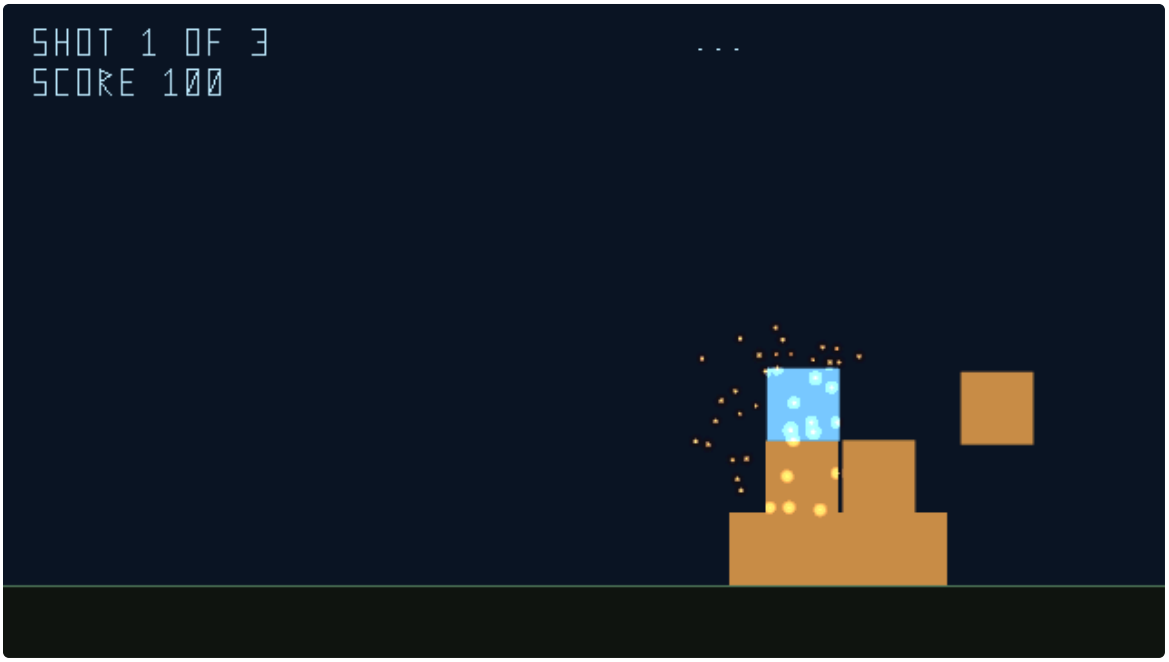


Figure: Crate Smash just after the first impact (run with `--demo 1`, which fires automatically so the screenshot shows action). The orange boxes are the pyramid, the blue box is the projectile, and the sparks are `pfx_burst` particles.

You can run it yourself with `ling run crate.ling`, or add `--demo 1` to watch a scripted first shot.

How it fits together

The structure mirrors the previous sections one for one.

State. `Game` is immutable data. `update` returns a *new* `Game` every frame, which is why each branch ends by constructing one. The only things that live outside the record are the physics world and the particle pool, which are global by nature.

`choose` and `match`. `update` is one big `match` on the phase, and each arm handles only its own phase. In `Flying(t, hit)` the arm names the two fields it needs. `Over()` and anything unknown fall through to `_ => g`, so adding a phase later cannot crash the loop, though you will want to handle it deliberately.

Rigid bodies. A new projectile body is added per shot and appended to `g.bodies`, and `floor_all` calls `rb_floor` on every body every frame, including old projectiles. That is the workaround for having no remove or teleport: you keep the list of everything you ever added.

The y flip lives in exactly two functions, `sx` and `sy`. Nothing else in the program knows that screen y is down. Keep conversions in one place and the y-up trap disappears.

Fixed time step. `dt` is a constant 0.016, the step the physics was tested with. Physics prefers constant steps to variable ones; on a very slow machine the game plays in slow motion rather than falling apart. A variable step with a cap, as in the animation chapter, is the alternative.

Rotation. Bodies are 3D boxes but we draw them in 2D. Since all forces here act in the x-y plane, a body only ever rotates about the z axis, and the quaternion `[x, y, z, w]` reduces to an angle: `2 * atan2(z, w)`. We rotate four corner offsets by that angle and hand the polygon to `draw_poly`.

Input to update. `update` takes `turn`, `push` and `fire`, never calling `key_down` itself. Called with `fire = true` and no window, it would launch a projectile, which makes it testable.

Effects. The first frame the projectile gets within 1.1 metres of a crate, `touched` flips and the burst and `audio_sfx` fire exactly once, because the `Flying` variant remembers `hit`. Remembering that an effect has played inside the state is how you get one-shot behaviour without assignment.

> **WARNING**
>
> The physics has no remove, and the game has no restart: three shots leave nine bodies in the world, and a second round would mean a second program run. If you want replayable rounds, give each round its own offset in x (the world is infinite sideways) and build a new tower there, or design the game around endless spawning of bodies that you simply stop drawing.

> **TRY IT**
>
> Add a fourth state, `Aiming` with wind: a constant sideways force applied with `rb_impulse` every frame during `Flying`, drawn as an arrow in the HUD. Which functions need to change, and which do not?

Extending it with soft bodies

For a very different feel, `soft_ball(x, y, z, radius)` creates a deformable ball of 86 nodes, `soft_step(id, dt, gravity)` advances it, `soft_bounce(id, floor_y, restitution)` bounces it and `soft_nodes(id)` returns the node positions. Unlike `rb_*`, the soft-body world is **y-down**, matching the screen, so you do not flip y. The two worlds are independent and use opposite conventions, a good reason to keep each in its own conversion function.

```
fn r2(x) { round(x * 100) / 100 }

bind start = do {
  bind ball = soft_ball(0, 0 - 3, 0, 1)
  println("nodes: ", len(soft_nodes(ball)) / 3)
  bind lowest = 0 - 99
  bind f = 0
  while f < 90 {
    soft_step(ball, 0.016, 9.8)
    soft_bounce(ball, 2, 0.5)
    bind c = soft_centroid(ball)
    bind lowest = max(lowest, c[1])
    bind f = f + 1
  }
  println("lowest centre reached: ", r2(lowest))
  println("final centre y: ", r2(soft_centroid(ball)[1]))
}
```

```
OUTPUT
nodes: 86
lowest centre reached: 1.3
final centre y: 0.29
```

The ball starts 3 units *above* the origin (negative y is up here), falls under +9.8 gravity, and the floor sits at y = 2. A rigid ball's centre could not go lower than y = 1, one radius above the floor. The soft ball's centre reaches 1.3, because the ball squashes on impact. After the bounce it is on its way back up, which the much smaller final y confirms.

Chapter summary

- `rb_*` is one global rigid-body world: `rb_add`, `rb_gravity`, `rb_impulse`, `rb_step`, `rb_floor` (call per body per frame), `rb_pos`, `rb_rot`. Bodies are unit cubes.
- **Physics y is up** (gravity -9.81, floor at 0). Screen y is down. Convert in exactly one place.
- There is no body removal, teleport or velocity getter. Keep a list of every body you add, and floor them all every frame.
- Model game phases with `choose`, keep the whole state in one record, and write `update(state, input, dt)` as a `match` on the phase.
- Gather input first, update second, draw third. Pass input into `update` as plain values.
- `pfx_burst`, `pfx_step`, `pfx_draw` give you particles in 3D space; with the default camera and `z = 4`, one unit is 40 pixels in a 640 by 360 window.
- `audio_blip` and `audio_sfx` synthesise effects without files; remember effects that should play once inside the state.
- `soft_*` bodies are y-down; `rb_*` bodies are y-up.

Exercises

1. Add a score bonus: 50 points per unused shot when the tower is fully cleared. Which function changes, and why does it not need a new variable?
2. Make the HUD show a power bar with `ui_bar` that fills with `g.power`.
3. Draw a trail: keep the last twelve projectile positions in the game record and draw them as fading discs.

4. Add a `Menu` phase before `Aiming` with a "press Space to start" prompt. What changes in `update` and `draw_hud`?

Answers

1. `knocked(g.tower)` counts crates; the score shown in `draw_hud` becomes `knocked * 100 + (if knocked == 6 { (3 - g.shots) * 50 } else { 0 })`. The score is computed from the current state each time, so nothing needs to be stored or updated, a general advantage of deriving values rather than tracking them.
2. `ui_bar(380, 54, 140, 10, g.power, 16)` inside the `Aiming()` arm of `draw_hud`. The bar widget takes the value and the maximum; set the colour with a trailing `r, g, b` if you want it to turn red at high power.
3. Add a `trail` list field to `Game`. In the `Flying` arm append the current `rb_pos` of the projectile (a two-number list) and drop the oldest when the list exceeds twelve, copying it into a new list since there is no pop. Draw each entry with `set_alpha(i / 12)`.
4. Add `Menu` to the `Phase` choose. In `update`, `Menu() => if fire { Game(Aiming(), ...) } else { g }`. In `draw_hud` add an arm that draws the prompt. Start the game in `Menu()`.

CHAPTER 8

Music Tools

What you'll learn

Ling has a built-in music toolkit: analysis functions that listen to an audio file (`music_*`), and a digital audio workstation engine (`daw_*`) that can build, edit and render music from code. This chapter uses both to make three tools: an *analyser* that reports tempo and onsets of a sound, a *generator* that composes and renders a melody, and a *step sequencer* with a window and a click-to-edit grid. Along the way you will export MIDI and read it back, and learn where the built-in analysis is trustworthy and where you should compute the answer yourself.

Everything but the final sequencer window runs headless: no sound card, no assets. We generate our own audio with `daw_*` and analyse it with `music_*`, so every number in this chapter comes from a real run.

The two halves: `music_*` and `daw_*`

`music_*` **consumes** audio. `music_load(path)` decodes a WAV, FLAC, OGG, MP3 or M4A file and returns a small integer handle (0, 1, 2, ...), or `-1` if the file cannot be read. With the handle you can ask for the duration, the estimated tempo and key, the times of note onsets, a beat grid, and (during playback) spectrum bands.

`daw_*` **produces** audio. It is a project-based engine: you create a project, add tracks, put clips on the tracks, put notes in the clips, and then either play the project through the speakers or *bounce* it, render it to a file, without needing an output device at all. The bounce is silent and synchronous, which is exactly what a test or a batch tool wants.

The two meet in the middle: `daw_bounce` writes a WAV; `music_load` reads it back.

> NOTE
>
> The default JIT backend swallows runtime errors raised inside `daw_*` calls. `daw_track_add("nope")` quietly does nothing under `ling run` and stops the program with an error under `ling run --interp`. While you are developing music code, run with `--interp` to see the mistakes, then switch back.

Analysing a sound

Start with the simplest generated sound: a two-bar loop of drum hits, one every half beat, at 120 BPM.

```
bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind drums = daw_track_add("drums")
  bind clip = daw_clip_add_midi(drums, 0, 8)
  bind step = 0
  while step < 16 {
    daw_note_add(drums, clip, step * 0.5, 0.25, 36, 0.9)
    bind step = step + 1
  }
  daw_bounce("loop.wav", 0, 8, 0.0, 16, "wav", 0.5)

  bind track = music_load("loop.wav")
  println("track handle: ", track)
  println("seconds: ", round(music_duration(track)))
  println("tempo: ", round(music_bpm(track)))
  println("onsets: ", len(music_onsets(track)))
  println("beats in grid: ", len(music_beat_grid(track)))
}
```

```
OUTPUT
track handle: 0
seconds: 4
tempo: 120
onsets: 15
beats in grid: 8
```

Step by step. `daw_new()` starts an empty project, `daw_set_bpm(120)` sets the tempo, and `daw_track_add("drums")` returns a track handle. `daw_clip_add_midi(track, start_beat, length_beats)` puts a clip on the track, here 8 beats (two bars) long, and `daw_note_add(track, clip, beat, length, key, velocity)` places a note *relative to the clip*. Key 36 is a kick drum. We add sixteen hits, one every half beat (two per beat at 120 BPM, so 0.25 seconds apart).

`daw_bounce(path, start_beat, end_beat, tail_seconds, bits, format, quality)` renders beats 0 to 8 to `loop.wav`. The result is exactly four seconds (8 beats at 2 beats per second), and the analyser agrees: 4 seconds, 120 BPM, 8 grid beats. It found 15 onsets, not 16, because the very first hit starts at the beginning of the file where there is nothing for the detector to compare with. That is the first lesson about built-in analysis: **it is an estimate, and edge cases at the start and end of a file are its weakest spot.**

Getting a second opinion from the onsets

You can check the tempo yourself from the onset times. If hits are evenly spaced, the average gap between them is the beat period, and BPM is `60 / gap` for beat-spaced hits. Here the hits are *eighth notes*, two per beat, so BPM is `30 / gap`. A helper that does this with the list returned by `music_onsets` is a short loop:

```
bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind drums = daw_track_add("drums")
  bind clip = daw_clip_add_midi(drums, 0, 8)
  bind step = 0
  while step < 16 {
    daw_note_add(drums, clip, step * 0.5, 0.25, 36, 0.9)
    bind step = step + 1
  }
  daw_bounce("loop.wav", 0, 8, 0.0, 16, "wav", 0.5)
  bind on = music_onsets(music_load("loop.wav"))

  bind total = 0
  bind i = 1
  while i < len(on) {
    bind total = total + (on[i] - on[i - 1])
    bind i = i + 1
  }
  bind gap = total / (len(on) - 1)
  println("mean gap between hits: ", round(gap * 1000) / 1000, " s")
  println("tempo if hits are eighth notes: ", round(30 / gap))
}
```

```
OUTPUT
mean gap between hits: 0.251 s
tempo if hits are eighth notes: 120
```

The same answer by a different route gives you confidence in both. In real tools, comparing two estimators is the cheapest sanity check you can build.

Where the built-in key detector falls short

`music_key(track)` returns a text such as `"A minor"`, estimated by chroma analysis. It is useful on full mixes of real recordings. On our simple synthesised melodies it was not reliable: four different arpeggios (in C, A minor and G, and a C major scale) all reported the same key. When a result cannot be trusted, compute it from the data you control. For generated music you know the notes, so a key estimate from them is exact. We will build one in the generator.

Building blocks for generated music

Before composing anything, we need a few small functions that every music program ends up wanting. None is built in, all are short.

```
fn note_name(m) {
  bind names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"]
  names[m % 12] + str(floor(m / 12) - 1)
}

// scales as semitone offsets from the root
fn major() { [0, 2, 4, 5, 7, 9, 11] }
fn minor() { [0, 2, 3, 5, 7, 8, 10] }

// the degree-th note (0 = root, 7 = the octave above) of a scale
fn scale_note(root, scale, degree) {
  bind octave = floor(degree / 7)
  root + octave * 12 + scale[degree % 7]
}

// a small deterministic random generator: reproducible music
fn next_seed(s) { (s * 75 + 74) % 65537 }

bind start = do {
  println(note_name(60), " ", note_name(69), " ", note_name(36))
  println(music_hz(69), " ", music_note_name(440))
  bind i = 0
  bind line = ""
  while i < 8 {
    bind line = line + note_name(scale_note(60, major(), i)) + " "
    bind i = i + 1
  }
  println(line)
  bind s = 1
  bind j = 0
  bind seq = ""
  while j < 5 {
    bind s = next_seed(s)
    bind seq = seq + str(s % 7) + " "
    bind j = j + 1
  }
  println(seq)
}
```

```
OUTPUT
C4 A4 C2
440 A4
C4 D4 E4 F4 G4 A4 B4 C5
2 0 3 6 3
```

Three points. `note_name` uses MIDI numbers, where 60 is middle C and every 12 is an octave. `music_hz(pitch)` converts a MIDI number to a frequency (69 gives 440 Hz) and `music_note_name(hz)` goes the other way. And `next_seed` is a tiny linear congruential generator. Ling's `rand()` takes no arguments and cannot be seeded, so a program that must produce the *same* music from the same input, for a test or for a "share this song" seed, needs its own. The multiplier is kept small so every intermediate value stays far below 2^53 where numbers are exact.

Project: a melody generator

Now compose. The generator walks a scale: from each note it takes a small random step (mostly up or down by one degree, sometimes a leap), picks a duration from a short list, and finishes on the root. It renders the result and reports on it. Every decision flows from one number, the seed.

```
fn note_name(m) {
  bind names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"]
  names[m % 12] + str(floor(m / 12) - 1)
}
fn major() { [0, 2, 4, 5, 7, 9, 11] }
fn scale_note(root, scale, degree) {
  bind octave = floor(degree / 7)
  root + octave * 12 + scale[degree % 7]
}
fn next_seed(s) { (s * 75 + 74) % 65537 }

// Returns a flat list: [degree0, length0, degree1, length1, ...]
fn compose(seed, count) {
  bind out = list_new()
  bind s = seed
  bind degree = 7
  bind i = 0
  while i < count {
    bind s = next_seed(s)
    bind move = s % 7
    bind step = match move {
      0 => -2,
      1 => -1,
      2 => -1,
      3 => 1,
      4 => 1,
      5 => 2,
      _ => 0
    }
    bind degree = clamp(degree + step, 0, 14)
    bind s = next_seed(s)
    bind len_beats = if s % 4 == 0 { 1 } else { 0.5 }
    bind out = list_push(out, degree)
    bind out = list_push(out, len_beats)
    bind i = i + 1
  }
  out
}
```

```
fn render(melody, root, bpm, path) {
  daw_new()
  daw_set_bpm(bpm)
  bind lead = daw_track_add("synth")
  bind total = 0
  bind i = 0
  while i < len(melody) {
    bind total = total + melody[i + 1]
    bind i = i + 2
  }
  bind clip = daw_clip_add_midi(lead, 0, total)
  bind at = 0
  bind i = 0
  while i < len(melody) {
    daw_note_add(lead, clip, at, melody[i + 1] * 0.9, scale_note(root, major(),
melody[i]), 0.8)
    bind at = at + melody[i + 1]
    bind i = i + 2
  }
  daw_bounce(path, 0, total, 0.5, 16, "wav", 0.5)
  total
}

bind start = do {
  bind melody = compose(7, 16)
  bind line = ""
  bind i = 0
  while i < len(melody) {
    bind line = line + note_name(scale_note(60, major(), melody[i])) + " "
    bind i = i + 2
  }
  println(line)
  bind beats = render(melody, 60, 110, "melody.wav")
  bind t = music_load("melody.wav")
  println("beats: ", beats, ", seconds: ", round(music_duration(t) * 10) / 10)
```

```
  println("estimated tempo: ", round(music_bpm(t)), " (composed at 110)")
  println("onsets found: ", len(music_onsets(t)), " (notes composed: 16)")
}
```

```
OUTPUT
D5 D5 C5 E5 F5 D5 C5 A4 B4 A4 G4 G4 F4 A4 B4 A4
beats: 9.5, seconds: 5.7
estimated tempo: 110 (composed at 110)
onsets found: 15 (notes composed: 16)
```

The melody is reproducible: run it again and you get the same line, and `compose(8, 16)` gives a different tune. The tempo estimate came back as exactly 110 here, but do not count on that for sparse or syncopated melodies: the detector likes a steady pulse. The onset count is again one short of the 16 notes, the same start-of-file edge case as before.

Reading the key from the notes

Since we have the notes, here is an exact key-finder, a few lines and no audio. It counts how often each of the twelve pitch classes occurs, then scores each candidate root against the major or minor scale by summing the counts of the scale's notes. The root with the best score wins; ties favour the major interpretation of the same notes.

```
fn major() { [0, 2, 4, 5, 7, 9, 11] }
fn minor() { [0, 2, 3, 5, 7, 8, 10] }

fn histogram(notes) {
  bind h = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  bind i = 0
  while i < len(notes) {
    bind pc = notes[i] % 12
    bind h = list_set(h, pc, h[pc] + 1)
    bind i = i + 1
  }
  h
}

fn score(h, root, scale) {
  bind total = 0
  bind k = 0
  while k < 7 {
    bind total = total + h[(root + scale[k]) % 12]
    bind k = k + 1
  }
  // reward the tonic and the fifth: they define the key
  total + h[root] + h[(root + 7) % 12]
}

fn guess_key(notes) {
  bind names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"]
  bind h = histogram(notes)
  bind best = -1
  bind best_name = ""
  bind r = 0
  while r < 12 {
    if score(h, r, major()) > best {
      bind best = score(h, r, major())
      bind best_name = names[r] + " major"
    }
    if score(h, r, minor()) > best {
      bind best = score(h, r, minor())
```

```
      bind best_name = names[r] + " minor"
    }
    bind r = r + 1
  }
  best_name
}

bind start = do {
  println(guess_key([60, 62, 64, 65, 67, 69, 71, 72]))
  println(guess_key([57, 60, 64, 57, 59, 62, 55, 59, 62, 57, 60, 64]))
  println(guess_key([67, 69, 71, 72, 74, 76, 78, 79]))
}
```

```
OUTPUT
C major
A minor
G major
```

Three answers: C major for the plain scale, A minor for a progression built around Am, G and C, and G major for a G scale with its F sharp. The reward for tonic and fifth is a rule of thumb, not music theory: change it and watch which ambiguous cases flip. That is the point of building it yourself.

MIDI: exporting and reading back

MIDI files hold notes, not sound, so they are small and editable everywhere. `daw_midi_export_project(path)` writes the current project; `music_midi_load(path)` loads a file as a silent source of notes; `music_midi_count` counts them; `music_midi_notes` returns a flat list `[time, key, time, key, ...]` and `music_midi_bars` a flat list `[time, key, duration, ...]` with durations. Times are in seconds.

```
fn note_name(m) {
  bind names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"]
  names[m % 12] + str(floor(m / 12) - 1)
}

bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind t = daw_track_add("synth")
  bind c = daw_clip_add_midi(t, 0, 4)
  daw_note_add(t, c, 0, 1, 60, 0.8)
  daw_note_add(t, c, 1, 1, 64, 0.8)
  daw_note_add(t, c, 2, 2, 67, 0.8)
  daw_midi_export_project("song.mid")

  bind m = music_midi_load("song.mid")
  println("notes: ", music_midi_count(m))
  bind bars = music_midi_bars(m)
  bind i = 0
  while i < len(bars) {
    println(note_name(bars[i + 1]), " starts at ", bars[i], " s for ", bars[i + 2], "
s")
    bind i = i + 3
  }
  println(file_exists("song.mid"))
}
```

```
OUTPUT
notes: 3
C4 starts at 0 s for 0.5 s
E4 starts at 0.5 s for 0.5 s
G4 starts at 1 s for 1 s
true
```

At 120 BPM a beat is half a second, so the note lengths in beats (1, 1, 2) come back as 0.5, 0.5 and 1 second. The round trip is faithful, which makes MIDI the right format for handing a generated tune to a notation program or another DAW. Feed `music_midi_bars` to a drawing loop and you have a piano-roll display or the notes for a rhythm game.

> TIP
>
> Rendering to WAV needs the whole sound engine; exporting MIDI needs only the notes. For a tool that must run on a server or inside a test, MIDI is the lighter artifact to generate and to check.

The step sequencer

A step sequencer is a grid. Each row is a sound, each column a sixteenth-note step in a bar, and a cell is on or off. It is the classic way to program drums, and it is a perfect Ling data structure: a list of rows, each a list of sixteen 0s and 1s.

The following program defines the pattern, renders two bars of it to a WAV, and analyses the result, the generator workflow applied to a sequencer.

```
// rows: kick, snare, hat; 16 steps each
fn pattern() {
  [
    [1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
  ]
}

fn row_key(r) { [36, 38, 42][r] }

fn hits(p) {
  bind n = 0
  bind r = 0
  while r < len(p) {
    bind s = 0
    while s < 16 {
      bind n = n + p[r][s]
      bind s = s + 1
    }
    bind r = r + 1
  }
  n
}

fn render(p, bpm, bars, path) {
  daw_new()
  daw_set_bpm(bpm)
  bind drums = daw_track_add("drums")
  bind clip = daw_clip_add_midi(drums, 0, bars * 4)
  bind bar = 0
  while bar < bars {
    bind r = 0
    while r < len(p) {
      bind s = 0
      while s < 16 {
        if p[r][s] == 1 {
          daw_note_add(drums, clip, bar * 4 + s * 0.25, 0.2, row_key(r), 0.9)
```

```
        }
        bind s = s + 1
      }
      bind r = r + 1
    }
    bind bar = bar + 1
  }
  daw_bounce(path, 0, bars * 4, 0.0, 16, "wav", 0.5)
}

bind start = do {
  bind p = pattern()
  println("hits per bar: ", hits(p))
  render(p, 120, 2, "beat.wav")
  bind t = music_load("beat.wav")
  println("seconds: ", round(music_duration(t)))
  println("tempo: ", round(music_bpm(t)))
  println("onsets: ", len(music_onsets(t)))
}
```

```
OUTPUT
hits per bar: 13
seconds: 4
tempo: 120
onsets: 15
```

A nested list indexed as `p[row][step]` is all the model you need. Changing a cell means rebuilding a row with `list_set`, and rebuilding the pattern with `list_set` again, which is a one-line helper we will write inside the app.

The numbers: the kick row has 3 hits, the snare 2 and the hat 8, so 13 hits per bar, which is what the program prints. The detector reports 15 onsets for two bars, not 26, and that is right: hits that land on the same step (a kick on a hat) are one event in time. The pattern occupies 8 distinct steps per bar, 16 in two bars, and the detector misses only the very first one, exactly as it did with the drum loop.

The window

Now the interface. A 16 by 3 grid of cells; click a cell to toggle it; a playhead sweeps across at the tempo; each active cell under the playhead triggers an `audio_blip` with a different pitch and waveform per row; a RENDER button writes the pattern to `beat.wav`. The model is the `pattern` list; the UI functions take it and return it, as in the UI chapter.

```
GUI
fn blank() {
  [
    [1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
  ]
}

fn toggled(p, r, s) {
  bind row = p[r]
  bind row = list_set(row, s, 1 - row[s])
  list_set(p, r, row)
}

fn cell_x(s) { 90 + s * 32 }
fn cell_y(r) { 70 + r * 44 }

fn label(x, y, px, s, r, g, b) {
  set_color(r, g, b)
  ui_text(x, y, px, s)
}

fn draw_grid(p, playhead) {
  bind names = ["KICK", "SNARE", "HAT"]
  bind r = 0
  while r < 3 {
    label(16, cell_y(r) + 11, 14, names[r], 150, 190, 215)
    bind s = 0
    while s < 16 {
      bind x = cell_x(s)
      bind y = cell_y(r)
      if p[r][s] == 1 {
        if s == playhead { set_color(255, 255, 255) } else { set_color(0, 210, 255) }
      } else {
        if s == playhead { set_color(60, 90, 120) } else { set_color(26, 38, 60) }
      }
      grad_rect(x, y, 28, 36, 0, 0, 0, 0, 0, 0, 1)
```

```
      if p[r][s] == 1 {
        if s == playhead { grad_rect(x, y, 28, 36, 255, 255, 255, 200, 230, 255, 1) }
else { grad_rect(x, y, 28, 36, 0, 210, 255, 0, 120, 200, 1) }
      } else {
        if s == playhead { grad_rect(x, y, 28, 36, 70, 100, 130, 50, 80, 110, 1) } else
{ grad_rect(x, y, 28, 36, 30, 44, 68, 22, 32, 52, 1) }
      }
      bind s = s + 1
    }
    bind r = r + 1
  }
}

fn clicked_cell(mx, my) {
  bind found = -1
  bind r = 0
  while r < 3 {
    bind s = 0
    while s < 16 {
      if mx >= cell_x(s) && mx < cell_x(s) + 28 && my >= cell_y(r) && my < cell_y(r) +
36 { bind found = r * 16 + s }
      bind s = s + 1
    }
    bind r = r + 1
  }
  found
}

fn play_step(p, s) {
  if p[0][s] == 1 { audio_blip(70, 0.14, "sine", 0.35) }
  if p[1][s] == 1 { audio_blip(220, 0.10, "noise", 0.25) }
  if p[2][s] == 1 { audio_blip(7000, 0.03, "noise", 0.12) }
}

bind start = do {
  open_window(640, 300, "Step Sequencer")
  bind p = blank()
  bind bpm = 120
  bind was_down = false
```

```
  bind last_step = -1
  while window_is_open() {
    // gather input
    bind down = mouse_down()
    bind clicked = down && !was_down
    bind was_down = down

    // update
    bind cell = if clicked { clicked_cell(mouse_x(), mouse_y()) } else { -1 }
    if cell >= 0 { bind p = toggled(p, floor(cell / 16), cell % 16) }
    bind playhead = floor(time_now() * bpm / 60 * 4) % 16
    if playhead != last_step { play_step(p, playhead) }
    bind last_step = playhead

    // draw
    fill(8, 12, 24)
    label(16, 20, 18, "STEP SEQUENCER", 190, 235, 255)
    draw_grid(p, playhead)
    bind bpm = clamp(ui_stepper(250, 232, 140, 28, bpm, 5), 60, 200)
    label(90, 238, 14, "TEMPO " + str(bpm), 150, 190, 215)
    present()
  }
}
```

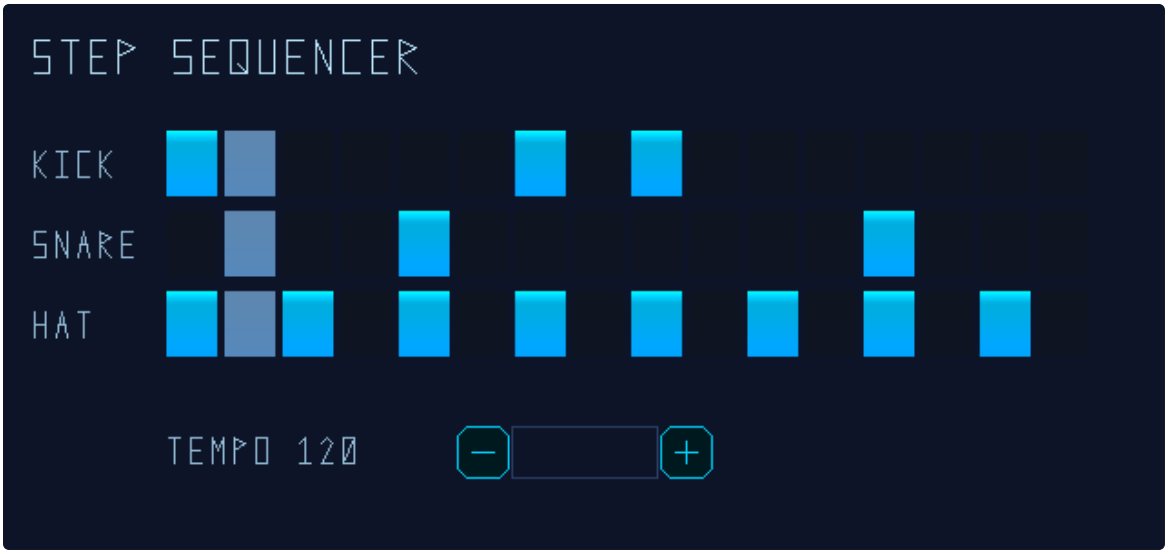


Figure: the sequencer with its default groove. The bright column is the playhead; filled cells are the active steps. Click any cell to toggle it.

The structure is the one from the UI and game chapters. The model is a nested list; `toggled` returns a new pattern; mouse input is reduced to a cell index by `clicked_cell`, which is plain arithmetic; and `playhead` is a pure function of time: `floor(time * bpm / 60 * 4) % 16` is "which sixteenth note are we in". Because the playhead is computed from the clock, not counted per frame, the groove keeps time even if frames are slow. The sound plays only on the frame the playhead changes (`last_step`), so each step fires once.

I verified that this program opens, draws a stable frame and exits cleanly, and the screenshot is a frame from that run. Mouse clicks and the actual sound of the blips need a person at a keyboard, so those I could not exercise automatically; the toggle logic (`toggled`, `clicked_cell`) is plain list code and behaves as the earlier headless sections show.

> **TRY IT**
>
> Add a fourth row, `OPEN HAT`, at key 46, and a RENDER button (see the UI chapter's `button` helper) that calls the `render` function from the previous section with the current pattern. Then `music_load` the result and show its onset count on screen.

Chapter summary

- `daw_new`, `daw_set_bpm`, `daw_track_add`, `daw_clip_add_midi`, `daw_note_add` build a project in code; `daw_bounce` renders it silently to a WAV; `daw_midi_export_project` writes a MIDI file.
- `music_load` returns a handle (or -1); `music_duration`, `music_bpm`, `music_onsets` and `music_beat_grid` analyse it. They are estimates: check them against your own arithmetic.
- `music_key` is unreliable on simple synthesised material. When you know the notes, compute the key from them.
- Write the small music helpers yourself: note names, scales as offset lists, and a seeded random generator, since `rand()` cannot be seeded.
- MIDI round-trips through `music_midi_load`, `music_midi_notes` and `music_midi_bars` (times in seconds).
- A step sequencer is a nested list indexed `p[row][step]`; the playhead is a function of the clock, not of the frame count.
- Under the default JIT, errors inside `daw_*` calls are swallowed; use `--interp` while developing.

Exercises

1. Make `compose` accept a scale argument and generate a tune in the minor scale. Check it with `guess_key` on the generated notes.
2. Write `swing(p)` that delays every odd step by 0.05 beat when rendering. Which line of `render` changes?
3. Use `music_onsets` on `beat.wav` to print the time of each hit in the first beat. How many onsets lie in the first 0.5 seconds?
4. Extend the sequencer's grid to 32 steps (two bars) without changing `play_step`. What must change in `cell_x` and in the playhead formula?

Answers

1. Pass `minor()` into `scale_note` through `render` and `compose` (compose itself works in degrees, so only `render` needs the scale). Collect the MIDI numbers into a list and give them to `guess_key`; a minor tune typically scores best as the relative or parallel minor, and ties are possible with short melodies.
2. The note start position: `bar * 4 + s * 0.25 + (if s % 2 == 1 { 0.05 } else { 0 })`. Everything else stays.
3. Loop over `music_onsets(t)` with a `while` and print entries below 0.5. Hits fall on steps 0 and 2 (times 0 and 0.25 s), so expect the first as the edge case the chapter described and about one reported onset.
4. Give `cell_x` a smaller cell width (a 640-pixel window fits 32 cells of 16 px) and change `% 16` in the playhead to `% 32`; the rows need 32 entries, and `clicked_cell`'s `r * 16 + s` becomes `r * 32 + s` along with the `floor(cell / 16)` decode.

CHAPTER 9

Testing and Debugging

What you'll learn

A program you cannot test is a program you are afraid to change. This chapter shows how to write tests in a language with no built-in assertions, what `lingfu test` really does (and the trap in it), how to use the interpreter as a free correctness checker for the default JIT backend, how to trace values with a pass-through logging helper, how to find out where the time goes, and how to read the error messages Ling produces. The chapter ends with a dictionary of the messages you will meet most often, each produced by a real run.

The through-line is honesty about the tools. Ling's default backend is forgiving to the point of silence, and its project test runner only notices a very particular kind of failure. Once you know the rules, you can build a reliable setup in about forty lines.

Two backends, two personalities

`ling run program.ling` uses the JIT compiler. `ling run --interp program.ling` uses the tree-walking interpreter. They are meant to behave identically, and for correct programs they do. For programs with mistakes they behave very differently, and that difference is the most useful debugging tool you have.

The JIT is fast and forgiving. A call to a function that does not exist yields `()` and carries on. A missing field is `()`. An index past the end of a list is empty. The interpreter is slow and strict: each of those is a runtime error with the line, a traceback and a hint. Here is the same typo run both ways.

`typo.ling`

```
FRAGMENT
bind start = do {
  bind x = sqroot(16)
  println("result: ", x)
}
```

```
OUTPUT
$ ling run typo.ling
result: ()
$ ling run --interp typo.ling
error[runtime]: unknown function 'sqroot'
 --> typo.ling
  traceback (deepest call last):
    0: start
  hint: check the spelling, or `use` the module that defines it
```

Under the JIT the program "works" and prints an empty result. Under the interpreter it stops at once and says exactly what is wrong. So the first rule of testing Ling is:

> **TIP**
>
> Before you ship, and in every test run, execute the program with `--interp`. A clean interpreter run catches misspelled functions, wrong argument counts, missing files, missing fields and bad indices that the JIT silently turns into `()`.

A few more differences you should know by heart, each with a run that proves it. Calling a function with too few arguments: the interpreter reports the missing parameter, the JIT returns `()`.

`args.ling`

```
FRAGMENT
fn add(a, b) { a + b }

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

```
OUTPUT
$ ling run args.ling
()
$ ling run --interp args.ling
error[runtime]: undefined: 'b'
 --> args.ling
  traceback (deepest call last):
    0: start
    1: add
  hint: check the spelling, or `use` the module that defines it
```

And the behaviour of `list_push`, where the JIT may modify the original list in place while the interpreter returns a new one. This one changes your program's *results*, not just its errors:

`alias.ling`

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

```
OUTPUT
$ ling run alias.ling
3 3
$ ling run --interp alias.ling
2 3
```

The interpreter's `2 3` is the intended answer: `a` is unchanged and `b` is the new list. The JIT's `3 3` shows aliasing. The fix is the habit you already have, always rebind (`bind a = list_push(a, 3)`), so that no code depends on whether the original changes. Two others from the language chapters: the `for` loop gives each iteration its own copy of the variables (so `bind total = total + i` inside `for` is lost on both backends), and `"a" < "b"` is `false` on the JIT but an error on the interpreter, because ordering is defined for numbers only.

Writing assertions

Ling has no `assert`. A test is a program that checks things and reports, so the job is a few helper functions. Put them in a library, with a prefix so that they never collide with your code, and load them from every test. The design follows from the language: functions cannot mutate caller variables, so each check *returns* 0 for pass and 1 for fail, and the test adds the results up.

`src/lib/testkit.ling`

FRAGMENT

```
// testkit: assertions that count failures

fn tt_check(label, condition) {
  if condition {
    println("  pass  ", label)
    0
  } else {
    println("  FAIL  ", label)
    1
  }
}

fn tt_eq(label, got, want) {
  if got == want {
    println("  pass  ", label)
    0
  } else {
    println("  FAIL  ", label, ": got ", got, ", wanted ", want)
    1
  }
}

fn tt_near(label, got, want) {
  if abs(got - want) < 0.000001 {
    println("  pass  ", label)
    0
  } else {
    println("  FAIL  ", label, ": got ", got, ", wanted ", want)
    1
  }
}

// Ends a test file. On failure it calls a function that does not exist:
// a deliberate runtime error, which makes `ling run --interp` exit with status 1.
fn tt_finish(failures) {
  if failures == 0 {
    println("all checks passed")
```

```
  } else {
    println(failures, " check(s) FAILED")
    tests_failed(failures)
  }
}
```

`tt_finish` is the interesting part. There is no `exit(code)` in Ling. The only portable way to make a script end with a failing exit status is to raise a runtime error, and the cleanest runtime error is calling a function that is not defined. On the interpreter that stops the program with status 1, which is exactly what a continuous-integration system or a shell script looks for. (On the JIT it does nothing, which is the point of the next section.)

A project with tests

Let us put the pieces together on a tiny statistics library. First create a project with `lingfu`, then add the library and its tests.

```
OUTPUT
$ lingfu new calc
~ Ling Manifest — Spirit Token Descends ~
Creating project: calc

✓ Spirit Realm created!
  cd calc
  lingfu build
  lingfu run
```

Here is the library, a first version with a bug in it that the tests are about to find:

`calc/src/lib/stats.ling`

```
FRAGMENT
// stats: small statistics helpers

fn st_sum(xs) {
  bind i = 0
  bind total = 0
  while i < len(xs) {
    bind total = total + xs[i]
    bind i = i + 1
  }
  total
}

fn st_mean(xs) {
  st_sum(xs) / len(xs)
}

fn st_max(xs) {
  bind best = 0
  bind i = 0
  while i < len(xs) {
    if xs[i] > best { bind best = xs[i] }
    bind i = i + 1
  }
  best
}
```

And the test. It lives in `tests/` and reaches the libraries by a relative path, since `use` paths are relative to the file that contains them. The test library is in `src/lib` rather than `tests/` for a reason we will meet in a moment.

`calc/src/lib/testkit.ling`

FRAGMENT

```
fn tt_check(label, condition) {
  if condition {
    println("  pass  ", label)
    0
  } else {
    println("  FAIL  ", label)
    1
  }
}

fn tt_eq(label, got, want) {
  if got == want {
    println("  pass  ", label)
    0
  } else {
    println("  FAIL  ", label, ": got ", got, ", wanted ", want)
    1
  }
}

fn tt_finish(failures) {
  if failures == 0 {
    println("all checks passed")
  } else {
    println(failures, " check(s) FAILED")
    tests_failed(failures)
  }
}
```

`calc/tests/test_stats.ling`

```
FRAGMENT
use "../src/lib/testkit.ling"
use "../src/lib/stats.ling"

bind test = do {
  bind failures = 0
  bind failures = failures + tt_eq("sum", st_sum([1, 2, 3]), 6)
  bind failures = failures + tt_eq("mean", st_mean([2, 4, 6]), 4)
  bind failures = failures + tt_eq("max of positives", st_max([3, 9, 4]), 9)
  bind failures = failures + tt_eq("max of negatives", st_max([0 - 5, 0 - 2, 0 - 9]), 0
- 2)
  tt_finish(failures)
}
```

Run it with the project runner, `lingfu test`:

```
OUTPUT
$ cd calc && lingfu test
Testing Ling project...
  test: ...\calc\tests\test.ling
starter test ok
  test: ...\calc\tests\test_stats.ling
  pass  sum
  pass  mean
  pass  max of positives
  FAIL  max of negatives: got 0, wanted -2
1 check(s) FAILED
✓ All tests passed!
```

Look closely at that output. The line `FAIL max of negatives: got 0, wanted -2` is there, the summary says `1 check(s) FAILED`... and the final line says **All tests passed!** That is the trap. `lingfu test` runs each test file with `ling run`, which uses the JIT, and it decides pass or fail from the process's exit status. Under the JIT, `tests_failed(...)` is an unknown function that quietly returns `()`, so the exit status is 0 and `lingfu test` reports success.

So `lingfu test` is useful for running everything and showing you the transcript, but you cannot trust its verdict for runtime failures. (It *will* fail on a parse error, since that stops the program before it starts.) To get a real verdict, run the file on the interpreter and look at the exit status:

```
OUTPUT
$ cd calc && ling run --interp tests/test_stats.ling
  pass  sum
  pass  mean
  pass  max of positives
  FAIL  max of negatives: got 0, wanted -2
1 check(s) FAILED
error[runtime]: unknown function 'tests_failed'
 --> tests/test_stats.ling
  traceback (deepest call last):
    0: start
    1: tt_finish
  hint: check the spelling, or `use` the module that defines it
(exit status 1)
```

Exit status 1 and the failing check named in the output: that is a test failure you can wire into a script. Now fix the bug. The error was in `st_max`: starting `best` at 0 is wrong when every number is negative. Start from the first element instead.

`calc/src/lib/stats.ling`

FRAGMENT

```
// stats: small statistics helpers

fn st_sum(xs) {
  bind i = 0
  bind total = 0
  while i < len(xs) {
    bind total = total + xs[i]
    bind i = i + 1
  }
  total
}

fn st_mean(xs) {
  st_sum(xs) / len(xs)
}

fn st_max(xs) {
  bind best = xs[0]
  bind i = 1
  while i < len(xs) {
    if xs[i] > best { bind best = xs[i] }
    bind i = i + 1
  }
  best
}
```

OUTPUT

```
$ cd calc && ling run --interp tests/test_stats.ling
  pass  sum
  pass  mean
  pass  max of positives
  pass  max of negatives
all checks passed
(exit status 0)
```

Green, with status 0. The edge cases are still your job: what does `st_mean([])` do? It divides by zero, which is `inf` on the interpreter and `()` on the JIT. A real library would return a `bad("empty list")` result, as in the errors chapter, and the test would check for it. Write the test first, watch it fail, then make it pass; the habit pays for itself the first time a refactor breaks something you had forgotten about.

> **WARNING**
>
> `lingfu test` runs *every* `.ling` file under `tests/`, and a file with no entry point fails with `no entry point`. That is why `testkit.ling` lives in `src/lib` and not beside the tests. Keep helper files out of the tests folder.

> **NOTE**
>
> Test files bind `test` rather than `start`, as the generated starter does. Any top-level `bind` of a `do` block is accepted as the entry. Tests that call `start` directly would run your whole program.

Tracing with a pass-through logger

When a result is wrong, the quickest tool is `println` in the middle of a computation. The trouble is that a print statement needs a line of its own, and inside an expression you have no line to spare. A *pass-through* helper prints a labelled value and returns it unchanged, so you can wrap any sub-expression without restructuring the code.

```
bind DEBUG = true

fn trace(label, v) {
  if DEBUG { println("[trace] ", label, " = ", v) }
  v
}

form Item { name, qty, price, }

fn line_total(it) {
  trace("qty", it.qty) * trace("price", it.price)
}

fn order_total(items) {
  bind i = 0
  bind sum = 0
  while i < len(items) {
    bind sum = sum + trace("line " + str(i), line_total(items[i]))
    bind i = i + 1
  }
  sum
}

bind start = do {
  bind order = [Item("tea", 3, 2.5), Item("kettle", 1, 24)]
  println("total: ", order_total(order))
  println(order[0])
}
```

```
OUTPUT
[trace] qty = 3
[trace] price = 2.5
[trace] line 0 = 7.5
[trace] qty = 1
[trace] price = 24
[trace] line 1 = 24
total: 31.5
Item { name: tea, qty: 3, price: 2.5 }
```

`trace("qty", it.qty) * trace("price", it.price)` computes the same value as without the traces. Printing a whole record (the last line) is often all the debugging you need, since a form prints with every field. When you are done, set `DEBUG = false` and the traces go quiet without being deleted.

> TIP
>
> Prefix debug output with something greppable, such as `[trace]`, so you can filter it from real output in a terminal.

Where does the time go?

Ling programs are usually fast enough, but "usually" is not a measurement. The clock is `time_now()`, seconds since the program started. Wrap the thing you want to time between two readings:

```
fn work(n) {
  bind i = 0
  bind s = 0
  while i < n {
    bind s = s + sqrt(i) * 2
    bind i = i + 1
  }
  s
}

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

bind start = do {
  bind t0 = time_now()
  bind r = work(1000000)
  bind t1 = time_now()
  println("a million loop steps: ", round((t1 - t0) * 1000), " ms")
  bind r2 = fib(25)
  bind t2 = time_now()
  println("fib(25) = ", r2, " in ", round((t2 - t1) * 1000), " ms")
}
```

OUTPUT

```
a million loop steps: 2 ms
fib(25) = 75025 in 0 ms
```

Now the same program on the interpreter, to see how different the two backends really are:

OUTPUT

```
$ ling run --interp perf.ling
a million loop steps: 291 ms
fib(25) = 75025 in 73 ms
```

On the machine used for this book, the JIT ran a million loop steps in a couple of milliseconds and the interpreter needed roughly two hundred and forty: about a hundred times slower. Your numbers will differ but the ratio will be similar. That is why `--interp` is a

checker, not a way to run your game, and why a program that is mysteriously slow should first be confirmed to be running on the JIT (the default). If a very large program takes a long time to compile, a lower optimisation level such as `--O1` is the usual knob.

Frame budgets and per-pixel work

A game at 60 frames per second has 16 milliseconds for everything: input, update, drawing and `present()`. Measure each stage with `time_now()` and print the slowest. Be especially suspicious of anything that touches every pixel.

```
GUI
bind start = do {
  open_window(320, 180, "Cost of pixels")
  bind n = 0
  while window_is_open() {
    bind n = n + 1
    if n > 3 { quit() }
    fill(0, 0, 0)

    bind t0 = time_now()
    bind y = 0
    while y < 180 {
      bind x = 0
      while x < 320 {
        set_color(x % 256, y, 128)
        draw_pixel(x, y)
        bind x = x + 1
      }
      bind y = y + 1
    }
    bind t1 = time_now()
    tex_noise(0, 0, 320, 180, 4, 1)
    bind t2 = time_now()

    println("57,600 draw_pixel calls: ", round((t1 - t0) * 1000), " ms; one tex_noise
over the same area: ", round((t2 - t1) * 1000), " ms")
    present()
  }
}
```

A run of this program on the book's machine printed about 10 ms for the per-pixel loop and about 2 ms for the single `tex_noise` call covering the same 57,600 pixels. Scale that up: at 1920 by 1080 the loop is 36 times more pixels, roughly 360 ms per frame, which is three frames per second. The rules that follow:

1. **Use area calls instead of pixel loops.** `fill`, `grad_rect`, `draw_poly`, `tex_*` over a rectangle and the `ui_*` widgets paint thousands of pixels per call.
2. **Do not call a `tex_*` generator in a pixel loop.** Each call is a whole-rectangle operation; calling it once per pixel multiplies its setup cost by the pixel count. Call it once per rectangle.
3. **Do not recompute what has not changed.** A static background does not need regenerating every frame.
4. **Hoist constants out of loops**, and avoid building lists inside inner loops (every `list_push` may allocate).
5. **Measure before you optimise.** Print timings for each stage; the slow part is almost never where you expect.

> NOTE
>
> There is no profiler. `time_now()` brackets plus `println` is the profiler. A tiny helper, `fn ms(since) { round((time_now() - since) * 1000) }`, makes each measurement one line.

A dictionary of error messages

Errors you will meet most, each produced by a real run of a one-line mistake. Parse errors are reported by both backends. Runtime errors are reported by the interpreter, so run with `--interp` to see them.

Assigning with `=` instead of rebinding with `bind`:

`e_assign.ling`

FRAGMENT

```
bind start = do {
  bind x = 1
  x = 2
  println(x)
}
```

```
OUTPUT
$ ling run e_assign.ling
error[parse]: unexpected token in expression: Eq
 --> e_assign.ling
```

Cause: there is no assignment statement. Write `bind x = 2`.

A missing closing brace:

`e_brace.ling`

```
FRAGMENT
bind start = do {
  println(1)
```

```
OUTPUT
$ ling run e_brace.ling
error[parse]: expected RBrace, got Eof
 --> e_brace.ling
```

Cause: `expected RBrace, got Eof` means a `{` was never closed; the parser ran out of file. The error points at the end of the file, not at the unmatched brace, so look for the last block you edited. An unterminated string literal gives the similar `expected RParen, got Eof`, since the string swallows the rest of the file.

A scientific-notation number, which Ling does not have:

`e_num.ling`

```
FRAGMENT
bind start = do {
  println(1.5e3)
}
```

```
OUTPUT
$ ling run --interp e_num.ling
error[runtime]: undefined: 'e3'
 --> e_num.ling
  traceback (deepest call last):
    0: start
  hint: check the spelling, or `use` the module that defines it
$ ling run e_num.ling
1.5()
```

Cause: `1.5e3` is lexed as the number `1.5` followed by the name `e3`. The interpreter complains about the undefined `e3`; the JIT prints `1.5` and an empty value. Write `1500`.

A range with a function call as its upper bound:

`e_range.ling`

```
FRAGMENT
bind start = do {
  bind xs = [1, 2, 3]
  for i in 0..len(xs) { println(i) }
}
```

```
OUTPUT
$ ling run --interp e_range.ling
error[runtime]: undefined: 'len'
 --> e_range.ling
  traceback (deepest call last):
    0: start
  hint: check the spelling, or `use` the module that defines it
```

Cause: the upper bound of `..` must be a single token or a parenthesised expression. `0..len(xs)` is read as `(0..len)(xs)`, and the interpreter then cannot find `len`. Write `0..(len(xs))`. On the JIT the same line does not fail, it silently loops zero times, which is worse.

A function used with the wrong number of arguments, a missing file, and ordering text:

`e_files.ling`

```
FRAGMENT
bind start = do {
  println(read_file("missing.txt"))
}
```

`e_cmp.ling`

```
FRAGMENT
bind start = do {
  println("a" < "b")
}
```

```
OUTPUT
$ ling run --interp e_files.ling
error[runtime]: read_file 'missing.txt': The system cannot find the file specified. (os
error 2)
 --> e_files.ling
  traceback (deepest call last):
    0: start
$ ling run --interp e_cmp.ling
error[runtime]: cannot convert 'a' to number
 --> e_cmp.ling
  traceback (deepest call last):
    0: start
```

Together with the ones we saw earlier in the chapter and the book, the summary table:

Message	Usual cause	Fix
`error[parse]: unexpected token in expression: Eq`	`x = 2` (no assignment)	`bind x = 2`
`error[parse]: expected RBrace, got Eof`	unclosed `{`	find the last edited block
`error[parse]: expected identifier, got LBrace`	typed last field in `form` with no trailing comma	`form P { x: number, y: number, }`
`error[parse]: unexpected token in expression: RParen`	a variable named like a keyword (`share`, `own`, `lend`, `move`, `copy`)	rename it

Message	Usual cause	Fix
`error[parse]: expected RBracket, got Comma`	a line that starts with `[` right after another statement is read as an index into it	`bind xs = [...]` first, then use `xs`
`error[runtime]: unknown function 'f'`	misspelt name; or calling a closure stored in a variable	fix spelling; use a named `fn`
`error[runtime]: undefined: 'b'`	too few arguments, an unknown variable, or a line starting with `(` right after a statement (read as a call on it)	check the call; bind the parenthesised value to a name
`error[runtime]: no field 'f' on Item {...}`	misspelt or missing record field	check the `form`
`error[runtime]: index 5 out of bounds`	list read past its end	compare with `len(xs)` first
`error[runtime]: cannot convert 'a' to number`	arithmetic or ordering on text	convert with `to_number` after validating
`error[runtime]: no entry point`	no `bind start = do { }` (a library file was run)	add or choose the right file
`read_file 'x': The system cannot find the file`	missing file	guard with `file_exists`

When the output is just wrong

No error, wrong answer: the hardest case. A short checklist, in the order that finds most bugs fastest:

1. **Run with `--interp`.** If the result changes, you have a JIT-silent mistake: a misspelt name, a missing field, an aliasing assumption.
2. **Look for a `for` loop that updates an outer variable.** It does not (a fresh scope each pass). Rewrite as `while`.
3. **Look for `==` between records or lists.** It never reports equality.
4. **Look for a missing rebind.** `list_push(xs, 1)` alone does nothing on the interpreter; it must be `bind xs = list_push(xs, 1)`.
5. **Look for a forgotten `present()`** (nothing appears) or **drawing order**: 3D shapes always land on top of 2D unless you call `flush_3d()`.
6. **Print the data.** `println(record)` and `println(list)` show everything.
7. **Shrink the program.** Delete code until the bug disappears, then put back the last thing you deleted.

Chapter summary

- The default JIT is forgiving and fast. `--interp` is strict and about a hundred times slower. Run your tests with `--interp`.
- There is no `assert`. Build `check` helpers that return failure counts, and finish each test file with a function that raises a runtime error on failure.
- `lingfu test` runs every `.ling` file under `tests/` with the JIT and judges by exit status, so runtime failures pass. Use it for the transcript and `ling run --interp tests/x.ling` for the verdict. Keep helper libraries out of `tests/`.
- A pass-through `trace(label, value)` lets you print inside an expression without restructuring.
- Time with `time_now()` brackets. Avoid per-pixel work; prefer area calls and widgets.
- Learn the common messages: parse errors name the token, runtime errors name the missing thing.

Exercises

1. Write `tt_near` tests for a `circle_area(r)` function and add them to the calc project's tests. Why is `tt_eq` the wrong tool?
2. Write `fn tt_summary(name, failures)` that prints one `name: ok` or `name: 2 failed` line. Then write a script file `tests/all.ling` that loads three test libraries and prints a combined summary. Which problem do you meet with `bind test`, and why?
3. Using only `time_now()`, measure how many `list_push` calls fit in 10 milliseconds on your machine, under the JIT and under `--interp`.
4. Take a program of your own, run it under `--interp`, and fix whatever it reports. Which class of error was it?

Answers

1. Floating-point results rarely compare exactly, and `tt_near` accepts a tolerance. `circle_area(1)` should be near `3.14159265`.
2. Each test file is its own program and `lingfu test` runs each file separately, so a combined run needs one file that `use`s all of the test code *as libraries*, which means the test bodies must be functions, not `bind test` blocks. Move each file's checks into `fn run_stats_tests() { ... failures }` and let `all.ling` call and add them.
3. Loop `while time_now() - t0 < 0.01`, counting iterations, and print the count. Expect orders of magnitude more iterations on the JIT than the interpreter.

4. Common results: a misspelt built-in name, a missing field, a `read_file` on a path relative to the wrong directory, or an index past the end of a list.

CHAPTER 10

Packaging and Shipping

What you'll learn

A program that runs on your machine is not yet a product. This chapter covers how to turn a Ling program into something other people can run: the `Ling.toml` manifest, the difference between `ling build` and `lingfu build`, app icons, the platforms you can target, bundling images and sounds with `includes` and `--pack`, and the WebAssembly route to a browser. It finishes with a release checklist and a small tool, written in Ling, that checks a project against it.

I need to be clear about one thing before we start. Building a native executable compiles generated Rust code with `cargo`, which needs a Rust toolchain and a working platform linker. In the environment where this book was verified, a native build was started but aborted by a local linker failure, and a WebAssembly build was started and began compiling but was not allowed to run to completion. So this chapter marks every statement as one of three kinds: **verified** (I ran it and show the output), **from the source** (I read how it works in the toolchain's code and documentation), or **not verified**. The release checker at the end is entirely verified, because it is a plain Ling program.

Two ways to run, three ways to build

Everything so far used `ling run` (and `lingfu run` for projects). Shipping needs a build:

- `ling build <file-or-folder> [options]` compiles a program, or a project folder, into a distributable. It produces one sub-folder per platform under an output folder (`dist` by default).
- `lingfu build` is the project manager's front end. For a project with a `Ling.toml` it calls `ling build <project> --aot` and passes your other options through.
- `ling run --wasm file.ling` builds to WebAssembly, serves it on a local web server and opens your browser. It is the quickest route to seeing a program on the web.

The options of `ling build` are small:

```
OUTPUT
ling build <file.ling | folder> [--out dir] [--platform list] [--icon file] [--pack] [--aot]
```

`--platform` takes a comma-separated list. The names are `web`, `win`, `lin`, `mac`, `all` (meaning all four) and some embedded targets (`macarm`, `pios`, `rpi`, `kernel`, `lingos`). Ask for a platform that does not exist and the tool says so. This is real output:

`main.ling`

```
FRAGMENT
bind start = do {
  println("hello from a packaged program")
}
```

```
OUTPUT
$ ling build main.ling --platform nope
Building 'main' v0.1.0 (Bin)
unknown platform 'nope' — use web|win|lin|mac|macarm|pios|kernel|rpi|lingos|all
```

That message is the authoritative list of platform names for this version: `web|win|lin|mac|macarm|pios|kernel|rpi|lingos|all`. (Verified.)

The manifest

A folder becomes a project when it has a manifest, `Ling.toml`. `lingfu new` writes one for you; for `ling build` you can write it by hand. Here is a manifest for a small game:

`game/Ling.toml`

FRAGMENT

```
[project]
name = "crate-smash"
version = "1.0.0"
entry = "src/start.ling"
icon = "art/icon.svg"
targets = ["win", "lin", "web"]
pack = true
ling-features = ["web"]

[includes]
"/art/*.png"
"/sound/*.wav"
```

The fields:

Key	Meaning
`name`	Project name; becomes the executable's name.
`version`	Version string (default `0.1.0`).
`entry`	The program to run first, relative to the manifest (default `main.ling`).
`icon`	Application icon, relative to the project root.
`targets`	Default platform list when you do not pass `--platform`.
`pack`	`true` embeds the included files inside the executable (same as `--pack`).
`ling-features`	Cargo features of the Ling runtime to build in. `["web"]` is required if the program uses the SQLite built-ins (`db_open` and friends), because those are feature-gated.
`[includes]`	Files to ship alongside the program, as glob patterns.

(From the source and the toolchain's documentation. The manifest syntax below is the part that is easy to get wrong.)

WARNING

`ling build` does not read your manifest with a real TOML parser. It scans it line by line for `key = value`. That has consequences: write one setting per line; put a space before the `=`; keep array values (`targets`, `ling-features`, `includes`) on a **single line**; and do not rely on `[sections]` to scope a key, because the keys are looked up wherever they appear. `lingfu run`, on the other hand, does parse real TOML. A manifest that is fine for one tool can therefore confuse the other, so keep to the simple style above.

The entry point

`lingfu new` creates the program at `src/start.ling`, but `ling build` assumes `main.ling` unless told otherwise. If you forget, the build stops immediately. Here is a manifest that names an entry that does not exist (verified):

`broken/Ling.toml`

```
FRAGMENT
[project]
name = "demo-game"
version = "1.0.0"
entry = "src/start.ling"
```

```
OUTPUT
$ ling build broken --platform win
error: entry '...\broken\src\start.ling' not found
```

The fix is to point `entry` at a file that exists, or to create it. For projects made with `lingfu new`, add `entry = "src/start.ling"` under `[spirit]` before you build.

Bundling your assets

Games need images, sounds, fonts and level files. There are two ways to ship them.

Alongside the executable (the default). The files that match `[includes]` are copied into the output folder, keeping their relative layout:

```
OUTPUT
dist/windows/
  crate-smash.exe
  art/hero.png
  sound/hit.wav
```

Your program loads them with their normal relative paths, `audio_sample_load("sound/hit.wav")`, `read_file("levels/one.txt")`. *(From the source.)*

Inside the executable (`--pack`). With `pack = true` in the manifest or `--pack` on the command line, the included files, and the entry program, are embedded in the executable, so you ship **one file**. At start-up they are unpacked into a temporary folder for that application

and the working directory is switched there, so `read_file`, `audio_sample_load` and `font_load` keep working with the same relative paths. *(From the source.)*

Patterns use `*` within one folder level, `**` across folders, and `?` for a single character. A bare folder name includes everything below it.

> **TIP**
>
> Develop with the loose files and a relative path, and test the packed build before you release. Anything that depends on the working directory being your source folder (a save file written next to the program, say) may behave differently once the program is unpacked somewhere temporary. Write save files to a location chosen explicitly, not to a relative path, if the player's progress matters.

> **NOTE**
>
> Relative paths are resolved from the *current directory of the running process*, not from the script's own folder. A program started by double-click from a different working directory can fail to find its files. This is true of your development runs as well, which is why consistent project layout matters.

Icons

On Windows, the executable carries an icon. The toolchain resolves it in this order:

1. `--icon <file>` on the command line.
2. `icon = "..."` in the manifest.
3. The built-in Ling logo.

Sources can be `.svg` (rendered faithfully, with gradients and blur, by the `resvg` renderer), `.png`, `.jpg`, `.bmp` (resized) or a ready-made `.ico`. Whatever you provide is packed into a multi-resolution `.ico` (16 to 256 pixels). If the icon you name does not exist the build prints a warning and falls back to the default logo, so check the build output.

The conversion is done by a crate called `ling-icon`, and you can use it directly. To make an `.ico` from an SVG yourself, the repository includes an example you run through Cargo:

```
OUTPUT
cargo run -p ling-icon --example svg2ico -- logo.svg out.ico
```

(From the documentation; I did not run the conversion here.)

A good icon is a simple, bold shape that is readable at 16 by 16 pixels. Draw it on a square canvas, give it a transparent or solid background deliberately, and view it small before you ship it.

Platforms

`ling build --platform win,lin,web` produces one folder per platform. What you need, and what to expect:

Target	Produces	Notes
`win`	`dist/windows/<name>.exe`	Needs a Rust toolchain with the MSVC linker.
`lin`	`dist/linux/<name>`	Needs a Linux target and its linker.
`mac`	`dist/macos/<name>`	Apple Silicon has its own name, `macarm`.
`web`	`dist/web/...`	A WebGL bundle built from the program compiled to WebAssembly.

The native targets generate a small Rust project and compile it with Cargo, so the first build can take minutes, and cross-compiling (building the Linux version on Windows, say) needs the matching Rust target and linker installed. In practice the dependable approach is to build each native platform **on that platform** (or on a CI machine of that type) and the web build from anywhere with the toolchain installed. *(From the source and documentation; native builds not verified here.)*

If `--platform` is omitted, the build uses the manifest's `targets`, and when that is empty the machine you are on plus `web`.

`--aot` compiles the program ahead of time to native machine code. Without it, the executable embeds the interpreter and your source. The AOT executable is faster to start and does not carry your source in readable form; the embedded-interpreter version is the more forgiving. `lingfu build` uses AOT unless you pass `--no-aot`. *(From the source.)*

The web target

For the browser, `ling run --wasm file.ling` and `ling build ... --platform web` compile the program to WebAssembly. In my verification I started both. The tool announced `Building WASM (this may take a minute on first run)` and began compiling the Ling runtime, which confirms the route works as far as that stage; I did not let it finish.

The browser is a different environment, and some builtins behave differently there. Documented differences you should design for:

- **Fonts:** `font_load` returns `-1` in a web build. Use the built-in `ui_text` stroke font or draw your own glyphs.
- **Gamepad:** `pad_pressed` behaves like `pad_button` in a browser (only a current snapshot is available), so press detection must be done by your own edge detection.
- **Modifier keys:** shift, ctrl and alt have no left/right variants in the browser.
- **Microphone and speech:** some of these built-ins return nothing on the web.
- **Files and databases:** the browser has no ordinary file system, and the SQLite and thread built-ins may be absent or limited.

The pattern is to isolate platform-dependent code in a few functions and test the web build separately; never assume that a feature that works on the desktop works in a page.

A release checklist

Everything above is easy to forget at the end of a long project. Here is a checklist, in the order you should work through it.

1. **Tests green on the interpreter.** `ling run --interp tests/...` for each test, exit status 0 (see the testing chapter).
2. **The program itself runs clean on `--interp`.** It reveals the errors that the JIT hides: misspelt functions, missing fields.
3. **The program runs on the default JIT** at a good frame rate, and you have timed the slowest frame.
4. **Version bumped** in `Ling.toml`, and the number shown somewhere in the program (a title screen or an about box).
5. **Entry point correct**: `entry = "src/start.ling"` or wherever the program is.
6. **Every asset listed** in `includes`, and every `includes` pattern matches something.
7. **Icon present** and readable at 16 pixels.
8. **Licences**: fonts, sounds and images you did not create have terms you comply with, and a credits screen if they require attribution.
9. **Build each target** on a machine of that type; **run the built artifact** on a clean machine that has never had Ling installed.
10. **Test** `--pack` if you use it: first start (unpacking) and later starts.
11. **Check the web build** separately for the differences listed above.
12. **Save and settings files** go somewhere writable and explicit, not a relative path.
13. **Write down the build command** you used, so the next release is a copy-paste.

Items 5 to 7 are mechanical, so let us automate them.

Project: a release checker

The checker is a Ling program. Given a project folder, it reads `Ling.toml`, finds the keys, and verifies that the entry file exists, that the icon exists, that a version is set, and that each file the `includes` mention exists. It uses only techniques from the data chapter: `read_file`, `split`, `trim`, `starts_with`, `file_exists`, and result-style messages.

`release_check.ling`

FRAGMENT

```
// release_check.ling - checks a project folder against the release checklist
// usage: ling run release_check.ling --dir path/to/project

fn manifest_value(text, key) {
  bind lines = split(text, "\n")
  bind found = ""
  bind i = 0
  while i < len(lines) {
    bind t = trim(lines[i])
    if starts_with(t, key + " =") || starts_with(t, key + "=") {
      bind eq = str_find(t, "=")
      bind found = trim(substr(t, eq + 1, len(t) - eq - 1))
      bind found = str_replace(found, "\"", "")
    }
    bind i = i + 1
  }
  found
}

fn problem(msg) {
  println("  PROBLEM  ", msg)
  1
}

fn ok_line(msg) {
  println("  ok       ", msg)
  0
}

fn check_file(dir, label, path) {
  if len(path) == 0 {
    problem(label + " is not set in Ling.toml")
  } else if file_exists(dir + "/" + path) {
    ok_line(label + ": " + path)
  } else {
    problem(label + " not found: " + path)
  }
```

```
}

fn include_paths(text) {
  // pulls the quoted patterns out of an [includes] section that has no globs
  bind out = list_new()
  bind lines = split(text, "\n")
  bind inside = false
  bind i = 0
  while i < len(lines) {
    bind t = trim(lines[i])
    if starts_with(t, "[") { bind inside = (t == "[includes]") }
    else if inside && starts_with(t, "\"") {
      bind p = str_replace(str_replace(t, "\"", ""), ",", "")
      if starts_with(p, "/") { bind p = substr(p, 1, len(p) - 1) }
      bind out = list_push(out, p)
    }
    bind i = i + 1
  }
  out
}

fn has_glob(p) { str_find(p, "*") >= 0 || str_find(p, "?") >= 0 }

fn check_project(dir) {
  bind manifest = dir + "/Ling.toml"
  if !file_exists(manifest) {
    problem("no Ling.toml in " + dir)
  } else {
    bind text = read_file(manifest)
    bind n = 0
    bind n = n + check_file(dir, "entry", manifest_value(text, "entry"))
    bind n = n + check_file(dir, "icon", manifest_value(text, "icon"))
    if len(manifest_value(text, "version")) == 0 {
      bind n = n + problem("version is not set")
    } else {
      bind n = n + ok_line("version: " + manifest_value(text, "version"))
    }
    bind inc = include_paths(text)
    bind i = 0
```

```
    while i < len(inc) {
      if has_glob(inc[i]) {
        println("  note     pattern not checked: ", inc[i])
      } else {
        bind n = n + check_file(dir, "include", inc[i])
      }
      bind i = i + 1
    }
    n
  }
}

bind start = do {
  bind dir = cli_arg("dir", ".")
  println("checking ", dir)
  bind problems = check_project(dir)
  if problems == 0 { println("ready to build") } else { println(problems, " problem(s)
to fix") }
}
```

Make a sample project with two deliberate faults: the icon is missing and one included file is absent.

`shipme/Ling.toml`

```
FRAGMENT
[project]
name = "crate-smash"
version = "1.0.0"
entry = "src/start.ling"
icon = "art/icon.svg"

[includes]
"/sound/hit.wav"
"/art/*.png"
```

`shipme/src/start.ling`

```
FRAGMENT
bind start = do {
  println("crate smash")
}
```

```
OUTPUT
$ ling run release_check.ling --dir shipme
checking shipme
  ok       entry: src/start.ling
  PROBLEM  icon not found: art/icon.svg
  ok       version: 1.0.0
  PROBLEM  include not found: sound/hit.wav
  note     pattern not checked: art/*.png
2 problem(s) to fix
```

Two problems found, one note about the glob pattern it cannot expand. Now we add the missing files and run it again. A placeholder icon and a sound file are enough for the check; in your project you would add the real ones.

`shipme/art/icon.svg`

```
FRAGMENT
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 64 64"><rect width="64"
height="64" rx="12" fill="#0a1020"/><rect x="14" y="14" width="36" height="36"
fill="#f0a040"/></svg>
```

`shipme/sound/hit.wav`

```
FRAGMENT
placeholder
```

```
OUTPUT
$ ling run release_check.ling --dir shipme
checking shipme
  ok       entry: src/start.ling
  ok       icon: art/icon.svg
  ok       version: 1.0.0
  ok       include: sound/hit.wav
  note     pattern not checked: art/*.png
ready to build
$ ling run --interp release_check.ling --dir shipme
checking shipme
  ok       entry: src/start.ling
  ok       icon: art/icon.svg
  ok       version: 1.0.0
  ok       include: sound/hit.wav
  note     pattern not checked: art/*.png
ready to build
```

The second run uses the interpreter, as the checklist demands of every tool you rely on: same answer on both backends. Notice how the checker is itself built from the pieces of this book: a `manifest_value` function in the style of the key-value store, `ok`-style messages that return a count, and a thin `start` that only wires command-line input to a function. You can extend it with every item of the checklist that a program can examine: a version string that matches the one in the program, no file larger than a limit in the assets folder (`file_size`), no `.wav` that fails `music_load`.

> **TRY IT**
>
> Add a check that every file under `sound/` loads (`audio_sample_load` returns -1 on failure, or `music_load` returns -1) and report the ones that do not. You will use `list_dir`.

Shipping on the web: a short recipe

1. Develop and test on the desktop.
2. Replace anything that uses `font_load`, the file system or threads with web-safe code, or guard it behind a small function you can swap.
3. Run `ling run --wasm yourgame.ling` to see it in your browser.

4. Run `ling build . --platform web` to produce `dist/web`, and upload that folder to any static web host.
5. Test the packaged page in more than one browser; check that audio starts after a click, since browsers block sound until the user interacts.

Chapter summary

- `ling build <file|folder>` makes a distributable for each `--platform` (`web`, `win`, `lin`, `mac`, and more). `lingfu build` is the project-level front end and uses ahead-of-time compilation by default.
- The manifest `Ling.toml` is read by `ling build` line by line: one `key = value` per line, a space before `=`, arrays on one line. Set `entry` explicitly.
- `icon` accepts SVG, PNG, JPG, BMP or ICO and is converted to a multi-resolution `.ico`; `--icon` overrides it.
- `includes` lists the files to ship; `--pack` (or `pack = true`) embeds them in the executable.
- Native builds need a Rust toolchain and the right linker for the target. Build each native platform on its own system.
- The web target uses WebAssembly and has real differences (no `font_load`, a different gamepad model, no ordinary files).
- A release is a checklist. Automate the mechanical items with a small Ling program.

Exercises

1. Extend the release checker to verify that `version` has three dot-separated numeric parts, using `split` and a digit test.
2. Write a `bump_patch(version)` function that turns `"1.0.9"` into `"1.0.10"`. Why do you not simply add 1 to the text?
3. Make the checker report the total size in bytes of the files named in `includes` (without globs).
4. Write the build commands for a Windows and a web release of the game as a table in a `RELEASING.md` file written by your program with `write_file`.

Answers

1. `split(v, ".")` returns text pieces; check `len(parts) == 3` and test each piece with the `all_digits` helper from the errors chapter.

2. Split into parts, convert the last with `to_number`, add 1, and join with `list_join(parts, ".")`. Adding to the text would concatenate or produce a number, because Ling's `+` works on both.
3. Call `file_size(path)` for each non-glob include that exists and sum with a `while` loop.
4. The commands are `ling build . --platform win --out dist` and `ling build . --platform web --out dist`. Build the text as a list of lines and write it with `write_file("RELEASING.md", list_join(lines, "\n"))`.

PART

Recipes

Copy-paste solutions for everyday problems.

CHAPTER 11

Core Recipes

Short, copy-paste snippets for the jobs you do in every program. Each one is a complete program: paste it into a file, run it with `ling run`, and compare with the output shown. Ling has no sort, reverse, map or dictionary built in, and closures cannot be passed to built-ins on the default JIT, so these recipes use plain functions and `while` loops. Remember the two scope rules that shape them: a `for` body cannot update an outer variable, so accumulate with `while`; and always rebind list results (`bind xs = list_push(xs, v)`).

Thousands separators

Turn `1234567` into `1,234,567` for reports and scores.

```
fn commas(n) {
  bind s = str(floor(n))
  bind out = ""
  bind i = 0
  while i < len(s) {
    if i > 0 && (len(s) - i) % 3 == 0 { bind out = out + "," }
    bind out = out + substr(s, i, 1)
    bind i = i + 1
  }
  out
}

bind start = do {
  println(commas(7))
  println(commas(1234))
  println(commas(1234567))
}
```

```
OUTPUT
7
1,234
1,234,567
```

Walk the digits left to right; before any digit whose distance from the end is a multiple of three, insert a comma. Handle negatives by formatting `abs(n)` and adding the sign.

Pad a number or text

Right-align a column of numbers, or left-align labels.

```
fn pad_left(t, w) { if len(t) < w { str_repeat(" ", w - len(t)) + t } else { t } }
fn pad_right(t, w) { if len(t) < w { t + str_repeat(" ", w - len(t)) } else { t } }
fn pad_zero(n, w) { if len(str(n)) < w { str_repeat("0", w - len(str(n))) + str(n) }
else { str(n) } }

bind start = do {
  println("[", pad_left("42", 6), "]")
  println("[", pad_right("name", 8), "]")
  println(pad_zero(7, 3), " ", pad_zero(1234, 3))
}
```

```
OUTPUT
[    42]
[name    ]
007 1234
```

`str_repeat(text, count)` builds the filler. A pad function never truncates: a value wider than the field is returned as is.

Centre text in a field

Centre a title over a rule in console output.

```
fn center(t, w) {
  bind gap = w - len(t)
  if gap <= 0 { t } else {
    bind left = floor(gap / 2)
    str_repeat(" ", left) + t + str_repeat(" ", gap - left)
  }
}

bind start = do {
  println("[", center("MENU", 12), "]")
  println("[", center("odd", 8), "]")
}
```

```
OUTPUT
[    MENU    ]
[  odd   ]
```

The extra space of an odd gap goes on the right.

Round to decimal places

`round` always rounds to a whole number. To keep two decimals, scale up, round, scale down.

```
fn round_to(x, places) {
  bind k = pow(10, places)
  round(x * k) / k
}

bind start = do {
  println(round_to(3.14159, 2))
  println(round_to(2.5, 0))
  println(round_to(1234.5678, 1))
  println(round_to(0.1 + 0.2, 3))
}
```

```
OUTPUT
3.14
3
1234.6
0.3
```

Floating-point noise (0.30000000000000004) disappears once you round. Numbers print without trailing zeros, so use the money recipe below for a fixed number of decimals.

Money with two decimals

Format a price as `$12.50` working in whole cents.

```
fn money(x) {
  bind cents = round(x * 100)
  bind rest = cents % 100
  "$" + str(floor(cents / 100)) + "." + (if rest < 10 { "0" } else { "" }) + str(rest)
}

bind start = do {
  println(money(12.5))
  println(money(0.07))
  println(money(1999.999))
}
```

```
OUTPUT
$12.50
$0.07
$2000.00
```

Rounding to cents first avoids results like `$12.4` or `$13.00` for 12.999.

Split, trim and count words

Break a sentence into words and count them.

```
fn words(text) {
  bind parts = split(trim(text), " ")
  bind out = list_new()
  bind i = 0
  while i < len(parts) {
    if len(parts[i]) > 0 { bind out = list_push(out, parts[i]) }
    bind i = i + 1
  }
  out
}

bind start = do {
  bind w = words("  the quick   brown fox  ")
  println(len(w), " words: ", list_join(w, "|"))
}
```

```
OUTPUT
4 words: the|quick|brown|fox
```

`split` keeps empty pieces for repeated separators, so filter them out.

Reverse a string

Ling has no string reverse; walk the characters backwards.

```
fn reverse_text(t) {
  bind out = ""
  bind i = len(t) - 1
  while i >= 0 {
    bind out = out + substr(t, i, 1)
    bind i = i - 1
  }
  out
}

fn is_palindrome(t) {
  bind clean = str_replace(str_lower(t), " ", "")
  clean == reverse_text(clean)
}

bind start = do {
  println(reverse_text("stressed"))
  println(is_palindrome("Never odd or even"))
  println(is_palindrome("Ling"))
}
```

```
OUTPUT
desserts
true
false
```

`substr(text, start, length)` with length 1 is the way to read one character.

Capitalise words

Upper-case the first letter of every word.

```
fn capitalise(w) {
  if len(w) == 0 { w } else { str_upper(substr(w, 0, 1)) + str_lower(substr(w, 1,
len(w) - 1)) }
}

fn title_case(t) {
  bind parts = split(t, " ")
  bind out = list_new()
  bind i = 0
  while i < len(parts) {
    bind out = list_push(out, capitalise(parts[i]))
    bind i = i + 1
  }
  list_join(out, " ")
}

bind start = do {
  println(title_case("the LORD of the rings"))
}
```

```
OUTPUT
The Lord Of The Rings
```

Split into words, fix each, and join again: the standard shape of every text transformation.

Count occurrences of a substring

How many times does a pattern appear?

```
fn count_of(text, needle) {
  bind n = 0
  bind i = 0
  while i <= len(text) - len(needle) {
    if substr(text, i, len(needle)) == needle {
      bind n = n + 1
      bind i = i + len(needle)
    } else {
      bind i = i + 1
    }
  }
  n
}

bind start = do {
  println(count_of("banana", "an"))
  println(count_of("aaaa", "aa"))
  println(count_of("hello", "z"))
}
```

```
OUTPUT
2
2
0
```

Skipping past each match counts non-overlapping occurrences (`aaaa` has two `aa`). Advance by 1 instead to count overlapping ones.

Wrap text to a width

Break a paragraph into lines no longer than a width, on word boundaries.

```
fn wrap(text, width) {
  bind parts = split(text, " ")
  bind lines = list_new()
  bind line = ""
  bind i = 0
  while i < len(parts) {
    if len(line) == 0 {
      bind line = parts[i]
    } else if len(line) + 1 + len(parts[i]) <= width {
      bind line = line + " " + parts[i]
    } else {
      bind lines = list_push(lines, line)
      bind line = parts[i]
    }
    bind i = i + 1
  }
  if len(line) > 0 { bind lines = list_push(lines, line) }
  lines
}

bind start = do {
  bind out = wrap("Ling is a small language with a big toolbox for graphics, sound and
games", 24)
  bind i = 0
  while i < len(out) {
    println("|", out[i], "|")
    bind i = i + 1
  }
}
```

```
OUTPUT
|Ling is a small language|
|with a big toolbox for|
|graphics, sound and|
|games|
```

A word longer than the width gets a line to itself rather than being broken. For pixel widths in a window, replace `len(...)` with `font_width` or the stroke-font estimate.

Sum, minimum, maximum and average

The basic statistics, written by hand because a `for` loop cannot accumulate.

```
fn sum_of(xs) {
  bind i = 0
  bind t = 0
  while i < len(xs) { bind t = t + xs[i]  bind i = i + 1 }
  t
}

fn max_of(xs) {
  bind best = xs[0]
  bind i = 1
  while i < len(xs) {
    if xs[i] > best { bind best = xs[i] }
    bind i = i + 1
  }
  best
}

fn min_of(xs) {
  bind best = xs[0]
  bind i = 1
  while i < len(xs) {
    if xs[i] < best { bind best = xs[i] }
    bind i = i + 1
  }
  best
}

bind start = do {
  bind xs = [4, 9, 2, 7]
  println(sum_of(xs), " ", min_of(xs), " ", max_of(xs), " ", sum_of(xs) / len(xs))
  println(list_max([3, 8, 1], 0), " ", list_min([3, 8, 1], 0))
}
```

```
OUTPUT
22 2 9 5.5
8 1
```

Built-ins `list_max(list, default)` and `list_min(list, default)` exist for numbers; the hand-written versions show the pattern for anything else (longest text, cheapest record).

Reverse a list

Build a new list from the end to the start.

```
fn reversed(xs) {
  bind out = list_new()
  bind i = len(xs) - 1
  while i >= 0 {
    bind out = list_push(out, xs[i])
    bind i = i - 1
  }
  out
}

bind start = do {
  println(reversed([1, 2, 3, 4]))
  println(len(reversed([])))
}
```

```
OUTPUT
[4, 3, 2, 1]
0
```

Never reverse in place: this returns a new list and leaves the original alone, which behaves the same on both backends.

Sort numbers

Insertion sort: simple, stable, plenty fast for lists up to a few thousand items.

```
fn insert_sorted(sorted, v) {
  bind out = list_new()
  bind placed = false
  bind i = 0
  while i < len(sorted) {
    if !placed && v < sorted[i] {
      bind out = list_push(out, v)
      bind placed = true
    }
    bind out = list_push(out, sorted[i])
    bind i = i + 1
  }
  if !placed { bind out = list_push(out, v) }
  out
}

fn sort_numbers(xs) {
  bind out = list_new()
  bind i = 0
  while i < len(xs) {
    bind out = insert_sorted(out, xs[i])
    bind i = i + 1
  }
  out
}

bind start = do {
  println(sort_numbers([5, 2, 9, 1, 5, 6]))
  println(sort_numbers([]))
}
```

```
OUTPUT
[1, 2, 5, 5, 6, 9]
[]
```

Each value is inserted into a result that is already sorted. For descending order, reverse afterwards or flip the `<`.

Sort records by a field

Order a list of records by one of their fields (here, score descending).

```
form Player { name, score, }

fn sort_by_score(ps) {
  bind out = list_new()
  bind i = 0
  while i < len(ps) {
    bind v = ps[i]
    bind next = list_new()
    bind placed = false
    bind j = 0
    while j < len(out) {
      if !placed && v.score > out[j].score {
        bind next = list_push(next, v)
        bind placed = true
      }
      bind next = list_push(next, out[j])
      bind j = j + 1
    }
    if !placed { bind next = list_push(next, v) }
    bind out = next
    bind i = i + 1
  }
  out
}

bind start = do {
  bind ranked = sort_by_score([Player("ann", 40), Player("bo", 95), Player("cy", 70)])
  bind i = 0
  while i < len(ranked) {
    println(i + 1, ". ", ranked[i].name, " ", ranked[i].score)
    bind i = i + 1
  }
}
```

```
OUTPUT
1. bo 95
2. cy 70
3. ann 40
```

Same insertion idea, comparing `v.score` with `out[j].score`. To sort by another field, change only the comparison.

Remove duplicates

Keep the first occurrence of every value, in order.

```
fn contains(xs, v) {
  bind i = 0
  bind found = false
  while i < len(xs) {
    if xs[i] == v { bind found = true }
    bind i = i + 1
  }
  found
}

fn unique(xs) {
  bind out = list_new()
  bind i = 0
  while i < len(xs) {
    if !contains(out, xs[i]) { bind out = list_push(out, xs[i]) }
    bind i = i + 1
  }
  out
}

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

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

`contains` compares with `==`, so this is for numbers and text, never records or lists.

Index of a value

Find where a value sits in a list, or -1.

```
fn index_of(xs, v) {
  bind at = -1
  bind i = len(xs) - 1
  while i >= 0 {
    if xs[i] == v { bind at = i }
    bind i = i - 1
  }
  at
}

bind start = do {
  println(index_of([10, 20, 30, 20], 20))
  println(index_of([10, 20, 30], 99))
}
```

```
OUTPUT
1
-1
```

Walking backwards while overwriting `at` leaves the *first* match, because the earliest index is written last.

Take, drop and concatenate

The slice operations Ling lacks.

```
fn take(xs, n) {
  bind out = list_new()
  bind i = 0
  while i < n && i < len(xs) { bind out = list_push(out, xs[i])  bind i = i + 1 }
  out
}

fn drop(xs, n) {
  bind out = list_new()
  bind i = n
  while i < len(xs) { bind out = list_push(out, xs[i])  bind i = i + 1 }
  out
}

fn concat(a, b) {
  bind out = take(a, len(a))
  bind i = 0
  while i < len(b) { bind out = list_push(out, b[i])  bind i = i + 1 }
  out
}

bind start = do {
  bind xs = [1, 2, 3, 4, 5]
  println(take(xs, 2), " ", drop(xs, 2), " ", concat(take(xs, 2), drop(xs, 4)))
}
```

```
OUTPUT
[1, 2] [3, 4, 5] [1, 2, 5]
```

`concat` starts from a copy made by `take`, so neither input is disturbed.

Make a numeric range as a list

Produce `[a, a+step, ...]` up to a limit.

```
fn range_list(from, to, step) {
  bind out = list_new()
  bind v = from
  while (step > 0 && v < to) || (step < 0 && v > to) {
    bind out = list_push(out, v)
    bind v = v + step
  }
  out
}

bind start = do {
  println(range_list(0, 5, 1))
  println(range_list(10, 0, -3))
  println(range_list(0, 1, 0.25))
}
```

```
OUTPUT
[0, 1, 2, 3, 4]
[10, 7, 4, 1]
[0, 0.25, 0.5, 0.75]
```

`for i in 0..n` is simpler when you only iterate, but a real list is handy for later indexing. Mind the rule that the range bound must be a single token: write `0..(len(xs))`.

Chunk a list and zip two lists

Group items in fixed-size rows, or pair two lists element by element.

```
fn chunk(xs, n) {
  bind out = list_new()
  bind row = list_new()
  bind i = 0
  while i < len(xs) {
    bind row = list_push(row, xs[i])
    if len(row) == n {
      bind out = list_push(out, row)
      bind row = list_new()
    }
    bind i = i + 1
  }
  if len(row) > 0 { bind out = list_push(out, row) }
  out
}

fn zip(a, b) {
  bind out = list_new()
  bind i = 0
  while i < len(a) && i < len(b) {
    bind out = list_push(out, [a[i], b[i]])
    bind i = i + 1
  }
  out
}

bind start = do {
  println(chunk([1, 2, 3, 4, 5, 6, 7], 3))
  println(zip(["a", "b", "c"], [1, 2]))
}
```

```
OUTPUT
[[1, 2, 3], [4, 5, 6], [7]]
[[a, 1], [b, 2]]
```

Lists of lists print as nested lists, and `row = list_new()` after pushing guarantees the saved row is never touched again.

Clamp, lerp and remap

Map a value from one range to another, the workhorse of UI and animation.

```
fn remap(x, a0, a1, b0, b1) {
  b0 + (clamp(x, a0, a1) - a0) / (a1 - a0) * (b1 - b0)
}

bind start = do {
  println(clamp(15, 0, 10), " ", lerp(10, 20, 0.25))
  println(remap(5, 0, 10, 100, 200))
  println(remap(15, 0, 10, 100, 200))
  println(remap(0.5, 0, 1, 255, 0))
}
```

```
OUTPUT
10 12.5
150
200
127.5
```

`clamp(x, lo, hi)` and `lerp(a, b, t)` are built in. `remap` combines them: normalise into 0..1, then scale into the target range. Use it for volume-to-bar-width, speed-to-colour and similar.

Greatest common divisor, primes, factorials

Three number-theory classics, short enough to type from memory.

```
fn gcd(a, b) {
  bind x = a
  bind y = b
  while y != 0 {
    bind t = y
    bind y = x % y
    bind x = t
  }
  x
}

fn is_prime(n) {
  bind ok = n >= 2
  bind d = 2
  while ok && d * d <= n {
    if n % d == 0 { bind ok = false }
    bind d = d + 1
  }
  ok
}

fn factorial(n) { if n <= 1 { 1 } else { n * factorial(n - 1) } }

bind start = do {
  println(gcd(48, 18), " ", gcd(17, 5))
  println(is_prime(97), " ", is_prime(91))
  println(factorial(10))
}
```

```
OUTPUT
6 1
true false
3628800
```

Numbers are 64-bit floats, so integers are exact up to about 9 quadrillion; `factorial(20)` is the last exact factorial.

Memoised Fibonacci with a list

Cache results in a list so each is computed once.

```
fn fib_table(n) {
  bind memo = [0, 1]
  bind i = 2
  while i <= n {
    bind memo = list_push(memo, memo[i - 1] + memo[i - 2])
    bind i = i + 1
  }
  memo
}

bind start = do {
  bind t = fib_table(50)
  println(t[10], " ", t[30], " ", t[50])
  println(len(t))
}
```

```
OUTPUT
55 832040 12586269025
51
```

With no map type, a list indexed by the argument is the memo table. Building it bottom-up (as here) avoids recursion entirely; for top-down memoisation, pass the table in and return both the value and the table.

Binary search

Find a value in a sorted list in about log2(n) steps.

```
fn binary_search(xs, v) {
  bind lo = 0
  bind hi = len(xs) - 1
  bind at = -1
  while lo <= hi && at < 0 {
    bind mid = floor((lo + hi) / 2)
    if xs[mid] == v { bind at = mid }
    else if xs[mid] < v { bind lo = mid + 1 }
    else { bind hi = mid - 1 }
  }
  at
}

bind start = do {
  bind xs = [2, 5, 8, 12, 16, 23, 38, 56]
  println(binary_search(xs, 23), " ", binary_search(xs, 2), " ", binary_search(xs, 7))
}
```

```
OUTPUT
5 0 -1
```

The `at < 0` test in the loop condition stands in for `break`, which Ling does not have.

Format seconds as minutes and seconds

Show a timer as `m:ss`.

```
fn clock_text(total) {
  bind s = floor(total)
  bind sec = s % 60
  str(floor(s / 60)) + ":" + (if sec < 10 { "0" } else { "" }) + str(sec)
}

bind start = do {
  println(clock_text(5))
  println(clock_text(65.9))
  println(clock_text(3600))
}
```

```
OUTPUT
0:05
1:05
60:00
```

`time_now()` gives seconds as a fraction; `floor` first so that 59.9 shows as 0:59 and not 1:00.

Stopwatch and elapsed time

Measure how long something took.

```
fn work() {
  bind i = 0
  bind s = 0
  while i < 200000 { bind s = s + i  bind i = i + 1 }
  s
}

bind start = do {
  bind t0 = time_now()
  bind r = work()
  bind ms = (time_now() - t0) * 1000
  println("result ", r)
  println("took under a second: ", ms < 1000)
}
```

```
OUTPUT
result 19999900000
took under a second: true
```

`time_now()` counts from program start; subtract two readings. The elapsed figure varies between runs, so tests should compare against a limit, as here.

Cooldown timer

Allow an action only every N seconds, without a variable you can change: pass the last time in and out.

```
fn ready(last, now, cooldown) { now - last >= cooldown }

bind start = do {
  bind last = 0 - 99
  bind fired = 0
  bind step = 0
  while step < 10 {
    bind now = step * 0.25
    if ready(last, now, 1) {
      bind fired = fired + 1
      bind last = now
    }
    bind step = step + 1
  }
  println("fired ", fired, " times in 2.25 simulated seconds")
}
```

```
OUTPUT
fired 3 times in 2.25 simulated seconds
```

In a real program `now` is `time_now()`. Drive the logic from a clock you pass in and you can test it with made-up times, as here.

Read a file line by line

Load a text file as a list of lines, skipping blanks.

```
fn read_lines(path) {
  bind out = list_new()
  if file_exists(path) {
    bind parts = split(read_file(path), "\n")
    bind i = 0
    while i < len(parts) {
      if len(trim(parts[i])) > 0 { bind out = list_push(out, trim(parts[i])) }
      bind i = i + 1
    }
  }
  out
}

bind start = do {
  write_file("demo.txt", "alpha\n\nbeta\r\ngamma\n")
  bind lines = read_lines("demo.txt")
  println(len(lines), " lines: ", list_join(lines, ","))
  println(len(read_lines("no-such-file.txt")))
  remove_file("demo.txt")
}
```

```
OUTPUT
3 lines: alpha,beta,gamma
0
```

Trimming each line also removes stray carriage returns from Windows-written files, and a missing file yields an empty list instead of an error.

Append a line to a file

Ling has no append mode; read, extend, write back.

```
fn append_line(path, line) {
  if file_exists(path) { write_file(path, read_file(path) + "\n" + line) } else {
write_file(path, line) }
}

bind start = do {
  remove_file("journal.txt")
  append_line("journal.txt", "first")
  append_line("journal.txt", "second")
  println(read_file("journal.txt"))
  remove_file("journal.txt")
}
```

```
OUTPUT
first
second
```

Fine for logs of up to thousands of lines. For heavy logging, collect lines in a list and write once at the end.

Pick a random integer and a random item

`rand()` returns a number from 0 up to but not including 1 and takes no arguments.

```
fn rand_int(lo, hi) { lo + floor(rand() * (hi - lo + 1)) }
fn pick(xs) { xs[floor(rand() * len(xs))] }

bind start = do {
  bind i = 0
  bind inside = true
  while i < 1000 {
    bind d = rand_int(1, 6)
    if d < 1 || d > 6 { bind inside = false }
    bind i = i + 1
  }
  println("all rolls between 1 and 6: ", inside)
  bind colour = pick(["red", "green", "blue"])
  println(len(colour) > 0)
}
```

```
OUTPUT
all rolls between 1 and 6: true
true
```

Random output differs on every run, so the program checks properties of the result instead of printing it. `hi - lo + 1` makes both ends inclusive.

Shuffle a list

Fisher-Yates: swap each position with a random earlier one.

```
fn shuffled(xs) {
  bind out = xs
  bind i = len(out) - 1
  while i > 0 {
    bind j = floor(rand() * (i + 1))
    bind a = out[i]
    bind out = list_set(out, i, out[j])
    bind out = list_set(out, j, a)
    bind i = i - 1
  }
  out
}

bind start = do {
  bind deck = [1, 2, 3, 4, 5, 6, 7, 8]
  bind s = shuffled(deck)
  bind total = 0
  bind i = 0
  while i < len(s) { bind total = total + s[i]  bind i = i + 1 }
  println(len(s), " cards, sum ", total, " (unchanged: 36)")
}
```

```
OUTPUT
8 cards, sum 36 (unchanged: 36)
```

`list_set` returns the changed list, so each swap is two rebinds. A shuffle must preserve the multiset, which the sum checks.

Seeded random numbers

`rand()` cannot be seeded. When you need the same sequence every run (a daily puzzle, a replay, a test) write a tiny generator.

```
fn next_seed(s) { (s * 75 + 74) % 65537 }
fn seeded_unit(s) { s / 65537 }

bind start = do {
  bind s = 12345
  bind line = ""
  bind i = 0
  while i < 5 {
    bind s = next_seed(s)
    bind line = line + str(floor(seeded_unit(s) * 100)) + " "
    bind i = i + 1
  }
  println(line)
}
```

```
OUTPUT
12 64 71 49 43
```

The multiplier is small enough that every product stays far below 2^53, where floating-point integers are exact. Same seed in, same numbers out.

A state machine with choose

Model a traffic light, a menu or an enemy's mood as a variant plus a function that returns the next state.

```
choose Light { Red, Green, Yellow }

fn next(l) {
  match l {
    Red() => Green(),
    Green() => Yellow(),
    Yellow() => Red(),
    _ => Red()
  }
}

fn seconds(l) {
  match l {
    Red() => 4,
    Green() => 3,
    _ => 1
  }
}

bind start = do {
  bind l = Red()
  bind i = 0
  while i < 5 {
    println(l.tag, " for ", seconds(l), "s")
    bind l = next(l)
    bind i = i + 1
  }
}
```

```
OUTPUT
Red for 4s
Green for 3s
Yellow for 1s
Red for 4s
Green for 3s
```

Keeping the transition in one function means the rules of the machine live in one place; adding a state means adding one arm to each `match`.

Stack and queue

Last-in-first-out and first-in-first-out with plain lists.

```
fn push(stack, v) { list_push(stack, v) }
fn top(stack) { stack[len(stack) - 1] }
fn pop(stack) {
  bind out = list_new()
  bind i = 0
  while i < len(stack) - 1 { bind out = list_push(out, stack[i])  bind i = i + 1 }
  out
}
fn dequeue(q) {
  bind out = list_new()
  bind i = 1
  while i < len(q) { bind out = list_push(out, q[i])  bind i = i + 1 }
  out
}

bind start = do {
  bind s = push(push(push(list_new(), 1), 2), 3)
  println(top(s))
  bind s = pop(s)
  println(top(s), " ", len(s))
  bind q = [10, 20, 30]
  println(q[0])
  bind q = dequeue(q)
  println(q[0], " ", len(q))
}
```

```
OUTPUT
3
2 2
10
20 2
```

Both are O(n) per operation because lists are copied, which is fine for hundreds of items. The top of a stack is its last element; the front of a queue is its first.

Parse a comma-separated line into numbers

Turn `"3, 4.5, 10"` into a list of numbers.

```
fn numbers(line) {
  bind parts = split(line, ",")
  bind out = list_new()
  bind i = 0
  while i < len(parts) {
    bind out = list_push(out, to_number(trim(parts[i])))
    bind i = i + 1
  }
  out
}

bind start = do {
  bind v = numbers("3, 4.5, 10")
  println(v, " sum ", v[0] + v[1] + v[2])
}
```

```
OUTPUT
[3, 4.5, 10] sum 17.5
```

`to_number` returns 0 for text that is not a number, so validate first if bad input is possible (see the errors chapter).

Convert a number to binary text

Print an integer in base 2 (or any base up to 16).

```
fn to_base(n, base) {
  if n == 0 { "0" } else {
    bind digits = "0123456789abcdef"
    bind out = ""
    bind v = floor(n)
    while v > 0 {
      bind out = substr(digits, v % base, 1) + out
      bind v = floor(v / base)
    }
    out
  }
}

bind start = do {
  println(to_base(10, 2), " ", to_base(255, 16), " ", to_base(255, 2), " ", to_base(0,
2))
  println(hex_to_number("ff"))
}
```

```
OUTPUT
1010 ff 11111111 0
255
```

Building the result by prepending each digit gives the correct order. `hex_to_number` is the built-in reverse for hexadecimal.

Roman numerals

A classic table-driven conversion.

```ling
fn roman(n) {
  bind values = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]
  bind marks = ["M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"]
  bind out = ""
  bind v = n
  bind i = 0
  while i < len(values) {
    while v >= values[i] {
      bind out = out + marks[i]
      bind v = v - values[i]
    }
    bind i = i + 1
  }
  out
}

bind start = do {
  println(roman(4), " ", roman(1994), " ", roman(2030))
}
```

```
OUTPUT
IV MCMXCIV MMXXX
```

Two parallel lists act as the lookup table. For each value, emit its mark as long as it fits.

A small key-value store

A lookup table with no map type: a list of pairs.

```
form Pair { key, value, }

fn find(store, key) {
  bind at = -1
  bind i = 0
  while i < len(store) {
    if store[i].key == key { bind at = i }
    bind i = i + 1
  }
  at
}

fn put(store, key, value) {
  bind i = find(store, key)
  if i < 0 { list_push(store, Pair(key, value)) } else { list_set(store, i, Pair(key,
value)) }
}

fn get(store, key, fallback) {
  bind i = find(store, key)
  if i < 0 { fallback } else { store[i].value }
}

bind start = do {
  bind s = list_new()
  bind s = put(s, "hp", 10)
  bind s = put(s, "mp", 4)
  bind s = put(s, "hp", get(s, "hp", 0) - 3)
  println(get(s, "hp", 0), " ", get(s, "mp", 0), " ", get(s, "xp", -1))
}
```

```
OUTPUT
7 4 -1
```

Updating a counter is `put(s, k, get(s, k, 0) + 1)`. For persistence see the data chapter.

CHAPTER 12

Graphics and UI Recipes

Copy-paste snippets for windows, drawing and interface work. Every recipe is a complete program fenced as `ling-gui`: it opens a window, so run it with `ling run` and close it with Escape. Each was run for this book with a frame counter that closed the window after a dozen frames. Conventions used throughout: the window is the frame loop `while window_is_open() { ... present() }`; screen y points down; colours are 0 to 255; `ui_text` y is the top of the line; and time comes from `time_now()`, never from counting frames. Variables are never called `move`, `share`, `copy`, `own` or `lend`: those are reserved words that wreck the parse.

Vertical and horizontal gradient fills

Paint a sky, a panel header or a bar with a smooth two-colour blend.

```ling-gui
bind start = do {
  open_window(480, 240, "Gradients")
  while window_is_open() {
    grad_rect(0, 0, 480, 240, 20, 30, 80, 250, 140, 90, 1)
    grad_rect(30, 30, 420, 30, 255, 80, 0, 0, 120, 255, 0)
    grad_rect(30, 90, 200, 120, 10, 10, 10, 90, 200, 140, 1)
    present()
  }
}
```

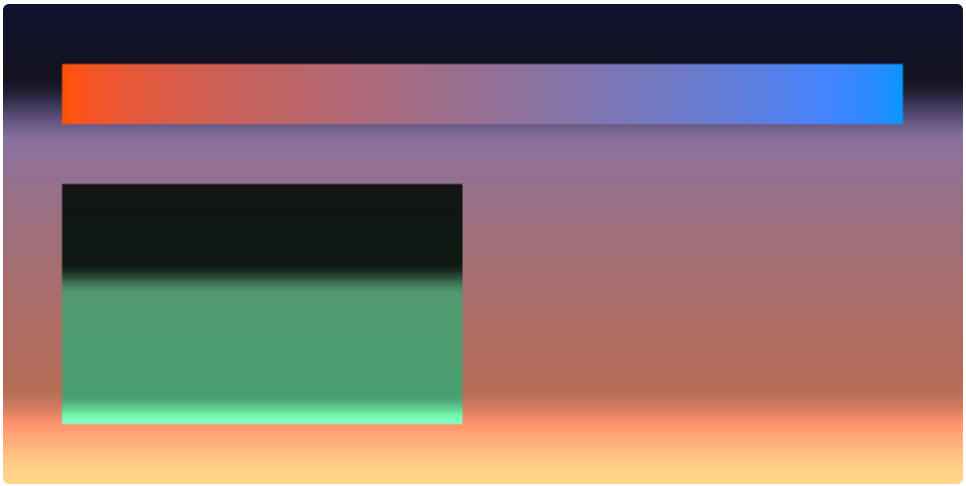


Figure: `grad_rect(x, y, w, h, r0, g0, b0, r1, g1, b1, dir)`*:* `dir` *0 blends left to right, anything else top to bottom.*

A flat rectangle is a gradient whose two colours are equal, which is how you get a filled rectangle: Ling has no separate `fill_rect`.

Filled rectangle helper

Draw a solid rectangle without remembering the gradient trick.

```
GUI
fn rect_fill(x, y, w, h, r, g, b) {
  grad_rect(x, y, w, h, r, g, b, r, g, b, 1)
}

fn rect_outline(x, y, w, h) {
  draw_line(x, y, x + w, y)
  draw_line(x + w, y, x + w, y + h)
  draw_line(x + w, y + h, x, y + h)
  draw_line(x, y + h, x, y)
}

bind start = do {
  open_window(320, 200, "Rectangles")
  while window_is_open() {
    fill(12, 16, 28)
    rect_fill(40, 40, 120, 80, 220, 150, 60)
    set_color(255, 255, 255)
    rect_outline(40, 40, 120, 80)
    present()
  }
}
```

Both helpers use only `grad_rect` and `draw_line`, so they work in every build and honour the current alpha.

Labelled button

A button that draws its own centred caption and returns the click.

```
GUI
fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

fn button(x, y, w, h, label) {
  bind clicked = ui_button(x, y, w, h)
  if ui_hot(x, y, w, h) { set_color(255, 255, 255) } else { set_color(190, 235, 255) }
  ui_text(x + (w - text_w(label, 14)) / 2, y + (h - 14) / 2, 14, label)
  clicked
}

bind start = do {
  open_window(360, 160, "Button")
  bind presses = 0
  while window_is_open() {
    fill(8, 12, 24)
    if button(40, 40, 140, 44, "PRESS ME") == 1 { bind presses = presses + 1 }
    set_color(120, 255, 160)
    ui_text(40, 110, 14, "PRESSES " + str(presses))
    present()
  }
}
```

`ui_button` returns `1` only on the frame the mouse goes down over it, so compare with `1` and the counter rises once per click. The stroke font has width `len * size * 0.86` less one gap, which makes centring arithmetic.

Progress ring with a percentage

A circular loading indicator with the number in the middle.

```
GUI
fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

bind start = do {
  open_window(240, 240, "Ring")
  while window_is_open() {
    bind pct = floor(fract((time_now() + 0.35) / 4) * 100)
    fill(8, 12, 24)
    ui_ring(120, 120, 70, pct, 100)
    set_color(190, 235, 255)
    bind label = str(pct) + "%"
    ui_text(120 - text_w(label, 28) / 2, 120 - 14, 28, label)
    present()
  }
}
```



Figure: `ui_ring(cx, cy, r, value, max)` with the percentage drawn in the middle.

`ui_ring(cx, cy, r, value, max)` takes a centre and radius. The label's top is `cy - size / 2`, and its left is `cx - width / 2`.

Tooltip on hover

Show a small bubble next to the cursor while it rests over a control.

```
GUI
fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

fn tooltip_for(x, y, w, h, text) {
  if ui_hot(x, y, w, h) {
    bind tw = text_w(text, 12) + 20
    ui_tooltip(mouse_x() + 14, mouse_y() + 18, tw, 28)
    set_color(220, 245, 255)
    ui_text(mouse_x() + 24, mouse_y() + 26, 12, text)
  }
}

bind start = do {
  open_window(360, 200, "Tooltip")
  while window_is_open() {
    fill(8, 12, 24)
    ui_button(40, 60, 120, 40)
    ui_button(190, 60, 120, 40)
    tooltip_for(40, 60, 120, 40, "SAVE YOUR GAME")
    tooltip_for(190, 60, 120, 40, "QUIT")
    present()
  }
}
```

Draw the tooltip last so it sits above everything else. Size the bubble from the text width so short and long tips both fit.

Toast notification

A message that slides in, waits, and fades out, driven by the time it was shown.

```
GUI
fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

fn toast(text, shown_at, now) {
  bind age = now - shown_at
  if age >= 0 && age < 3 {
    bind slide = if age < 0.3 { (1 - age / 0.3) * 40 } else { 0 }
    bind fade = if age > 2.4 { (3 - age) / 0.6 } else { 1 }
    set_alpha(fade)
    ui_panel(20 - slide, 20, text_w(text, 14) + 32, 40, 8)
    set_color(220, 245, 255)
    ui_text(36 - slide, 33, 14, text)
    set_alpha(1)
  }
}

bind start = do {
  open_window(420, 160, "Toast")
  bind shown_at = 0.0
  while window_is_open() {
    bind now = time_now() + 0.5
    fill(8, 12, 24)
    toast("GAME SAVED", shown_at, now)
    present()
  }
}
```

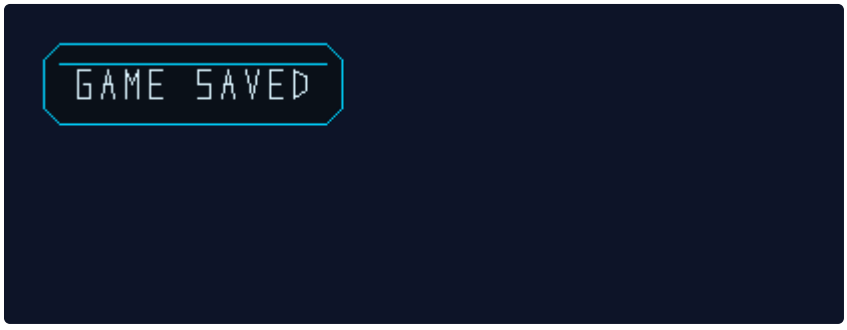


Figure: the toast a half second after it appears. `shown_at` is set to `time_now()` whenever something happens; the draw function turns age into position and opacity.

The whole animation is a pure function of `age`, so there is no timer variable to update. Trigger a new toast by setting `shown_at = time_now()`; to queue several, keep a list of `(text, time)` records.

Drag and drop

Pick up a box with the mouse, carry it, and drop it. The state is "which box, if any, is held, and where the grip was".

```
GUI
form Box { x, y, w, h, }

fn inside(b, mx, my) { mx >= b.x && mx < b.x + b.w && my >= b.y && my < b.y + b.h }

bind start = do {
  open_window(480, 240, "Drag")
  bind box = Box(80, 80, 90, 60)
  bind held = false
  bind grip_x = 0
  bind grip_y = 0
  while window_is_open() {
    bind mx = mouse_x()
    bind my = mouse_y()
    if mouse_down() {
      if !held && inside(box, mx, my) {
        bind held = true
        bind grip_x = mx - box.x
        bind grip_y = my - box.y
      }
    } else {
      bind held = false
    }
    if held { bind box = Box(mx - grip_x, my - grip_y, box.w, box.h) }

    fill(8, 12, 24)
    if held { set_color(255, 220, 120) } else { set_color(0, 210, 255) }
    grad_rect(box.x, box.y, box.w, box.h, 0, 120, 180, 0, 210, 255, 1)
    present()
  }
}
```

Record the offset between the cursor and the box's corner when you grab it (`grip_x`, `grip_y`), so the box does not jump to the cursor. Release when the button goes up. Dragging a list of boxes adds an index in place of `held`.

Minimap

Show the whole world in a small corner box, scaled down.

```
GUI
fn map_x(wx) { 330 + wx / 1000 * 120 }
fn map_y(wy) { 20 + wy / 600 * 72 }

bind start = do {
  open_window(480, 300, "Minimap")
  while window_is_open() {
    bind t = time_now() + 1
    bind px = 500 + 400 * sin(t * 0.5)
    bind py = 300 + 200 * cos(t * 0.7)
    fill(8, 12, 24)
    set_color(120, 170, 200)
    draw_disc(240 + (px - 500) * 0.2, 180 + (py - 300) * 0.2, 12)

    ui_panel(326, 16, 128, 80, 6)
    set_color(80, 110, 140)
    draw_disc(map_x(200), map_y(120), 3)
    draw_disc(map_x(850), map_y(480), 3)
    set_color(255, 220, 90)
    draw_disc(map_x(px), map_y(py), 3)
    present()
  }
}
```

Two small functions convert world coordinates to minimap pixels: divide by the world size, multiply by the box size, add the box origin. Draw static landmarks in one colour and the player in another.

Screenshot hotkey

Save the window to a PNG when F12 is pressed.

```
GUI
bind start = do {
  open_window(320, 180, "Screenshot")
  while window_is_open() {
    fill(10, 14, 28)
    set_color_hsl(fract(time_now() * 0.1) * 360, 80, 55)
    draw_disc(160, 90, 40)
    present()
    if key_pressed("f12") { screenshot("hotkey") }
  }
}
```

Call `screenshot(tag)` after `present()`; it writes `screenshots/ss_<time>_<tag>_<w>x<h>.png` in the current directory. The same call, scheduled from a frame counter, is how this book captured its own figures.

FPS counter

A smoothed frames-per-second readout.

```
GUI
bind start = do {
  open_window(320, 180, "FPS")
  bind last = time_now()
  bind fps = 60
  while window_is_open() {
    bind now = time_now()
    bind dt = max(now - last, 0.0001)
    bind last = now
    bind fps = fps + (1 / dt - fps) * 0.1
    fill(10, 14, 28)
    set_color(120, 255, 160)
    ui_text(12, 12, 14, "FPS " + str(round(fps)))
    present()
  }
}
```

The raw `1 / dt` jitters wildly, so blend a tenth of each new reading into the running value. That is an exponential moving average, the cheapest smoother there is.

Screen shake

Jolt everything on screen after an impact, fading to calm.

```gui
fn shake(now, hit_at, strength) {
  bind age = now - hit_at
  if age < 0 || age > 0.6 { [0, 0] } else {
    bind k = (1 - age / 0.6) * strength
    bind offset = [sin(age * 70) * k, cos(age * 90) * k]
    offset
  }
}

bind start = do {
  open_window(320, 200, "Shake")
  bind hit_at = 0.2
  while window_is_open() {
    bind now = time_now()
    bind off = shake(now, hit_at, 10)
    fill(10, 14, 28)
    set_color(255, 180, 70)
    draw_disc(160 + off[0], 100 + off[1], 40)
    set_color(40, 60, 90)
    draw_line(0 + off[0], 160 + off[1], 320 + off[0], 160 + off[1])
    if key_pressed("space") { bind hit_at = now }
    present()
  }
}
```

Apply the same offset to everything you draw. Note the `bind offset = [...]` line before the result: a line that *begins* with `[` (or `(`) directly after another statement is read as an index (or a call) into that statement (`k[...]`) and fails with `expected RBracket, got Comma`, so give a list result a name first. The strength falls linearly with age; two sine waves at unrelated frequencies make it look random without calling `rand()`.

Parallax background

Layers that scroll at different speeds create depth.

```
GUI
fn hills(offset, base, amp, r, g, b) {
  set_color(r, g, b)
  bind x = 0
  while x < 480 {
    bind h = base - amp * sin((x + offset) * 0.02) - amp * 0.5 * sin((x + offset) *
0.047)
    draw_line(x, h, x, 260)
    bind x = x + 2
  }
}

bind start = do {
  open_window(480, 260, "Parallax")
  while window_is_open() {
    bind t = time_now() * 40 + 200
    grad_rect(0, 0, 480, 260, 40, 60, 130, 255, 170, 120, 1)
    hills(t * 0.2, 150, 22, 90, 80, 140)
    hills(t * 0.5, 185, 26, 60, 60, 110)
    hills(t, 225, 24, 25, 30, 60)
    present()
  }
}
```

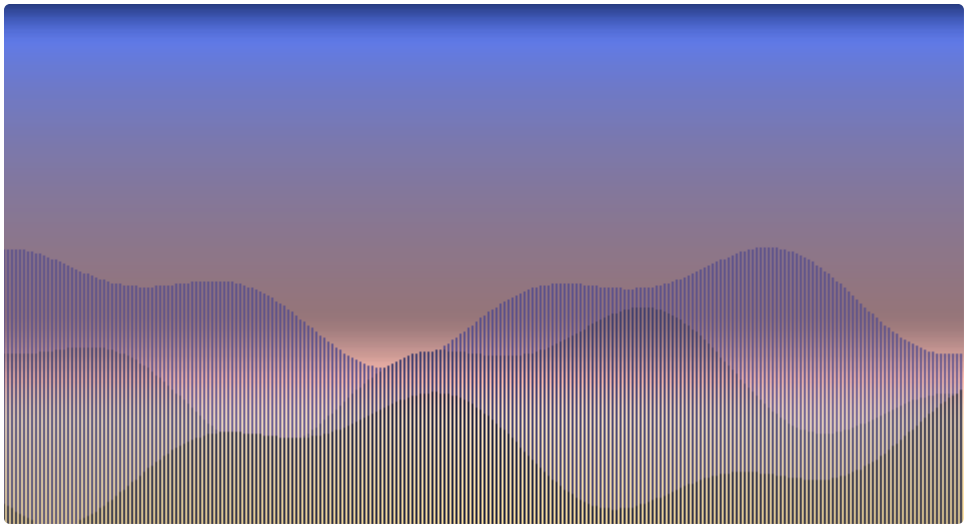


Figure: far layers scroll slowly, near layers fast, all from one clock.

One scroll value, multiplied by 0.2, 0.5 and 1 for far, middle and near layers. Draw the farthest layer first.

Smooth follow with a spring

Make something chase a moving target with a natural ease-in and slight overshoot.

```
GUI
bind start = do {
  open_window(400, 240, "Follow")
  bind px = 200
  bind py = 120
  bind vx = 0
  bind vy = 0
  bind last = time_now()
  while window_is_open() {
    bind now = time_now()
    bind dt = min(now - last, 0.05)
    bind last = now
    bind tx = 200 + 140 * cos(now * 1.3)
    bind ty = 120 + 80 * sin(now * 1.9)
    bind rx = spring_to(px, vx, tx, 120, 14, dt)
    bind ry = spring_to(py, vy, ty, 120, 14, dt)
    bind px = rx[0]
    bind vx = rx[1]
    bind py = ry[0]
    bind vy = ry[1]
    fill(10, 14, 28)
    set_color(255, 255, 255)
    draw_circle(tx, ty, 12)
    set_color(0, 210, 255)
    draw_disc(px, py, 10)
    present()
  }
}
```

`spring_to(pos, vel, target, stiffness, damping, dt)` returns `[new_pos, new_vel]`. Keep both numbers for each axis. Use it for cameras, trailing tails and cursors.

Wobbling and breathing

Idle motion for menus and characters, using the Anima drivers.

```
GUI
bind start = do {
  open_window(320, 200, "Idle")
  while window_is_open() {
    bind t = time_now()
    fill(10, 14, 28)
    set_color(255, 190, 80)
    draw_disc(110, 100 + wobble(t, 1.2, 8, 0), 28 + 6 * breathe(t, 0.5, 1))
    set_color(120, 220, 255)
    draw_disc(210, 100 + wobble(t, 1.2, 8, 0.25), 28 + 6 * breathe(t, 0.5, 1))
    present()
  }
}
```

`wobble(t, freq, amp, phase)` is a sine with a phase offset, `breathe(t, rate, depth)` a gentle in-and-out. Two copies a quarter-cycle apart look like a pair of bobbing characters.

Tile map from text

Describe a level as a list of strings and draw it.

GUI

```
fn tile_color(c) {
  match c {
    "#" => [90, 80, 110],
    "." => [26, 34, 52],
    "~" => [40, 90, 170],
    "*" => [255, 210, 80],
    _ => [0, 0, 0]
  }
}

fn draw_map(rows, size) {
  bind y = 0
  while y < len(rows) {
    bind x = 0
    while x < len(rows[y]) {
      bind c = tile_color(substr(rows[y], x, 1))
      grad_rect(x * size, y * size, size - 1, size - 1, c[0], c[1], c[2], c[0], c[1],
c[2], 1)
      bind x = x + 1
    }
    bind y = y + 1
  }
}

bind start = do {
  open_window(400, 240, "Tiles")
  bind level = [
    "###########",
    "#........#",
    "#..~~~...#",
    "#..~~~.*.#",
    "#........#",
    "###########"
  ]
  while window_is_open() {
    fill(0, 0, 0)
    draw_map(level, 40)
```

```
    present()
  }
}
```

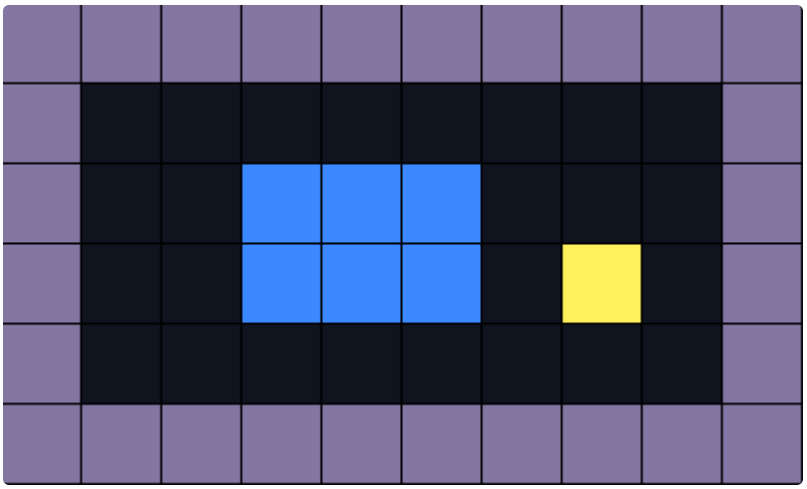


Figure: six strings become a level. Each character indexes a colour.

Edit levels in a plain text file and load them with `split(read_file("level.txt"), "\n")`. The same strings drive collision: `substr(rows[ty], tx, 1) == "#"` is a wall test.

Typewriter text

Reveal a message one character at a time.

```
GUI
bind start = do {
  open_window(480, 120, "Typewriter")
  bind message = "WELCOME TO THE SECTOR. STAY ALERT."
  while window_is_open() {
    bind shown = min(len(message), floor(time_now() * 18))
    fill(8, 12, 24)
    set_color(120, 255, 160)
    ui_text(20, 50, 16, substr(message, 0, shown))
    present()
  }
}
```

The number of visible characters is a function of time: 18 per second here. Restart the effect by subtracting the time the message began.

Fade in and out

A full-screen fade for scene changes, using alpha.

```
GUI
bind start = do {
  open_window(320, 200, "Fade")
  while window_is_open() {
    bind t = fract(time_now() / 4)
    bind a = if t < 0.5 { 1 - t * 2 } else { (t - 0.5) * 2 }
    fill(40, 90, 160)
    set_color(255, 255, 255)
    draw_disc(160, 100, 50)
    set_alpha(a)
    grad_rect(0, 0, 320, 200, 0, 0, 0, 0, 0, 0, 1)
    set_alpha(1)
    present()
  }
}
```

Draw the scene, then a black rectangle with alpha from 1 down to 0 (fade in) or 0 up to 1 (fade out). Always restore `set_alpha(1)`.

Pen-coloured rectangles and a rainbow

Fill a rectangle with the current pen colour, and sweep the hue around the wheel with `set_color_hsl`.

```
GUI
fn pen_rect(x, y, w, h) {
  draw_poly([x, y, x + w, y, x + w, y + h, x, y + h])
}

bind start = do {
  open_window(360, 160, "Pen rect")
  while window_is_open() {
    fill(10, 14, 28)
    bind i = 0
    while i < 12 {
      set_color_hsl((i * 30 + time_now() * 60) % 360, 85, 55)
      pen_rect(20 + i * 27, 40, 24, 80)
      bind i = i + 1
    }
    present()
  }
}
```

`draw_poly(points)` takes a flat list `[x0, y0, x1, y1, ...]` and fills with the pen colour, which `set_color_hsl(hue, saturation, lightness)` sets (hue in degrees). Note the difference from `grad_rect`, which ignores the pen and takes its own colours.

Rotate a polygon

Spin any shape around its centre by transforming its corner points.

```
GUI
fn rotated_square(cx, cy, size, angle) {
  bind pts = list_new()
  bind k = 0
  while k < 4 {
    bind a = angle + k * 1.5707963 + 0.7853982
    bind pts = list_push(pts, cx + cos(a) * size)
    bind pts = list_push(pts, cy + sin(a) * size)
    bind k = k + 1
  }
  pts
}

bind start = do {
  open_window(320, 200, "Spin")
  while window_is_open() {
    fill(10, 14, 28)
    set_color(255, 120, 90)
    draw_poly(rotated_square(160, 100, 50, time_now()))
    set_color(255, 220, 120)
    draw_poly(rotated_square(160, 100, 24, 0 - time_now() * 2))
    present()
  }
}
```

Each corner sits at `angle + k * 90 degrees` (plus 45 so the square starts flat) at radius `size`. Change the 4 and the step to get triangles, hexagons and stars.

Pie menu slices

A circular menu with the hovered slice highlighted.

```
GUI
fn slice_points(cx, cy, r, a0, a1) {
  bind pts = [cx, cy]
  bind steps = 10
  bind i = 0
  while i <= steps {
    bind a = a0 + (a1 - a0) * i / steps
    bind pts = list_push(pts, cx + cos(a) * r)
    bind pts = list_push(pts, cy + sin(a) * r)
    bind i = i + 1
  }
  pts
}

bind start = do {
  open_window(300, 300, "Pie")
  bind n = 6
  while window_is_open() {
    bind mx = mouse_x() - 150
    bind my = mouse_y() - 150
    bind ang = atan2(my, mx)
    bind hover = floor(((ang + 6.2831853) % 6.2831853) / (6.2831853 / n))
    if sqrt(mx * mx + my * my) > 110 || sqrt(mx * mx + my * my) < 20 { bind hover = -1
}
    if cli_arg("demo", "0") == "1" { bind hover = 1 }
    fill(8, 12, 24)
    bind i = 0
    while i < n {
      if i == hover { set_color(0, 210, 255) } else { set_color_hsl(220 + i * 12, 60,
28) }
      draw_poly(slice_points(150, 150, 100, i * 6.2831853 / n + 0.03, (i + 1) *
6.2831853 / n - 0.03))
      bind i = i + 1
    }
    present()
  }
}
```



Figure: a pie menu with the second slice active.

The hovered slice is the angle of the cursor (`atan2(dy, dx)`) divided by the slice size, with a dead zone near the centre and a limit at the rim. The `--demo 1` line only forces a slice on for the screenshot, where no mouse exists; remove it in your program.

Rotating 3D object (orbit view)

Orbit around a 3D scene by rotating the scene about its pivot, which is simpler than moving the camera and always keeps the scene centred.

```
GUI
fn turn_x(x, z, a) { x * cos(a) - z * sin(a) }
fn turn_z(x, z, a) { x * sin(a) + z * cos(a) }

bind start = do {
  open_window(480, 320, "Orbit")
  while window_is_open() {
    bind a = time_now() * 0.8 + 0.6
    fill(10, 14, 28)
    cube(0, 0, 5.5, 1.1, 1.1, 1.1, 0, a, 0, 0)
    bind i = 0
    while i < 6 {
      bind b = i * 1.0471976
      bind px = 2.4 * cos(b)
      bind pz = 2.4 * sin(b)
      sphere(turn_x(px, pz, a), 0 - 0.6, 5.5 + turn_z(px, pz, a), 0.4, 0.4, 0.4, 0, 0,
0, 0, 16, 10)
      bind i = i + 1
    }
    present()
  }
}
```

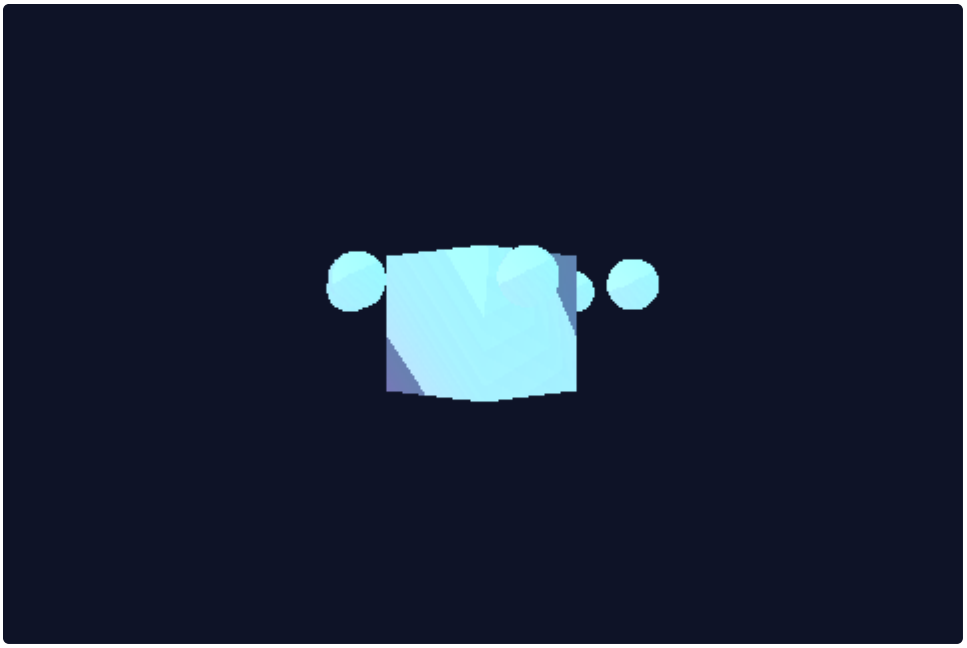


Figure: six spheres circling a cube. The scene turns, the camera never moves.

`turn_x` and `turn_z` rotate a point about the vertical axis through the scene centre (here at z = 5.5 in front of the camera). The cube turns with `ry = a`. Add a slow bob (`0 - 0.6 + 0.3 * sin(a)`) for life. If you want the user to drag-rotate, replace `time_now()` with an angle that grows with `mouse_dx()` after `capture_mouse()`.

Depth ordering: 2D under 3D

3D shapes are queued and always land on top of 2D drawing. Flush the 3D queue when you want 2D drawn over it.

```
GUI
bind start = do {
  open_window(320, 200, "Layers")
  while window_is_open() {
    fill(10, 14, 28)
    set_color(255, 120, 90)
    draw_disc(160, 100, 70)
    cube(0, 0, 6, 0.8, 0.8, 0.8, 0.4, time_now(), 0, 0)
    flush_3d()
    set_color(120, 255, 160)
    ui_text(12, 12, 14, "HUD ABOVE THE CUBE")
    present()
  }
}
```

Without `flush_3d()` the cube would be drawn after (and so over) the text. With it, the cube lands where it was queued and the HUD drawn next covers it. Use it before every HUD pass.

Tabbed panel

Tabs with captions, and different contents per tab.

```
GUI
fn text_w(s, px) { len(s) * px * 0.86 - px * 0.24 }

fn tab_captions(x, y, w, names) {
  bind each = w / len(names)
  bind i = 0
  while i < len(names) {
    set_color(190, 235, 255)
    ui_text(x + i * each + (each - text_w(names[i], 12)) / 2, y + 8, 12, names[i])
    bind i = i + 1
  }
}

bind start = do {
  open_window(420, 220, "Tabs")
  bind tab = 1
  while window_is_open() {
    fill(8, 12, 24)
    ui_panel(16, 16, 388, 188, 8)
    bind tab = ui_tabs(30, 30, 360, 28, 3, tab)
    tab_captions(30, 30, 360, ["VIDEO", "AUDIO", "INPUT"])
    set_color(150, 190, 215)
    match tab {
      0 => ui_text(40, 90, 14, "RESOLUTION AND QUALITY"),
      1 => ui_text(40, 90, 14, "VOLUME AND DEVICES"),
      _ => ui_text(40, 90, 14, "KEYS AND PADS")
    }
    present()
  }
}
```



Figure: `ui_tabs` draws the strip; the program draws captions and the active page.

`ui_tabs` returns the active index, so one `match` on it picks the page. Divide the strip's width by the tab count to centre each caption.

CHAPTER 13

Audio and Game Recipes

Snippets for sound, music and game logic. Programs fenced `ling` run headless and show their real output; programs fenced `ling-gui` open a window (and, where they play sound, need an audio device to be heard). Game logic is written as pure functions of state, so most of it can be shown without a window. Two reminders: the rigid-body world (`rb_*`) has y pointing up while the screen has y pointing down, and `rand()` cannot be seeded, so recipes that must be repeatable bring their own generator.

MIDI note to frequency table

Convert MIDI numbers to hertz and note names for tuning displays and synth code.

```
fn note_name(m) {
  bind names = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"]
  names[m % 12] + str(floor(m / 12) - 1)
}

fn hz(m) { 440 * pow(2, (m - 69) / 12) }

bind start = do {
  for m in [48, 57, 60, 69, 72] {
    println(m, " ", note_name(m), " ", round(hz(m) * 100) / 100, " Hz")
  }
  println(music_hz(60), " ", music_note_name(440))
}
```

```
OUTPUT
48 C3 130.81 Hz
57 A3 220 Hz
60 C4 261.63 Hz
69 A4 440 Hz
72 C5 523.25 Hz
261.62554931640625 A4
```

Every semitone multiplies the frequency by the twelfth root of two. The built-ins `music_hz(pitch)` and `music_note_name(hz)` do the same conversions; writing your own `hz` is useful when you want a different tuning, such as A at 432.

Scales and chords

Build scales and triads from semitone offsets.

```
fn major() { [0, 2, 4, 5, 7, 9, 11] }
fn minor() { [0, 2, 3, 5, 7, 8, 10] }

fn scale_notes(root, scale, count) {
  bind out = list_new()
  bind i = 0
  while i < count {
    bind out = list_push(out, root + floor(i / 7) * 12 + scale[i % 7])
    bind i = i + 1
  }
  out
}

fn triad(root, scale, degree) {
  bind notes = scale_notes(root, scale, 15)
  bind chord = [notes[degree], notes[degree + 2], notes[degree + 4]]
  chord
}

bind start = do {
  println(scale_notes(60, major(), 8))
  println(scale_notes(57, minor(), 8))
  println(triad(60, major(), 0), " ", triad(60, major(), 3), " ", triad(60, major(),
4))
}
```

```
OUTPUT
[60, 62, 64, 65, 67, 69, 71, 72]
[57, 59, 60, 62, 64, 65, 67, 69]
[60, 64, 67] [65, 69, 72] [67, 71, 74]
```

Degree 0 is the root; stacking degrees 0, 2 and 4 gives the triad, so the I, IV and V chords of C major are C-E-G, F-A-C and G-B-D.

Amplitude envelope (ADSR)

Shape a note's loudness over time: attack, decay, sustain, release.

```
fn adsr(t, held, a, d, s, r) {
  if t < 0 { 0 }
  else if t < a { t / a }
  else if t < a + d { 1 - (1 - s) * (t - a) / d }
  else if t < held { s }
  else if t < held + r { s * (1 - (t - held) / r) }
  else { 0 }
}

bind start = do {
  bind t = 0
  while t <= 1.01 {
    println(round(t * 100) / 100, " ", round(adsr(t, 0.6, 0.1, 0.2, 0.5, 0.3) * 100) /
100)
    bind t = t + 0.1
  }
}
```

```
OUTPUT
0 0
0.1 1
0.2 0.75
0.3 0.5
0.4 0.5
0.5 0.5
0.6 0.5
0.7 0.33
0.8 0.17
0.9 0
1 0
```

Multiply a waveform by this value, or drive a `ui_bar` with it. `held` is how long the key stays down; after that the release tail plays.

Render a drum pattern to a file and measure it

Generate audio with no files and no sound card, then analyse it.

```
bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind drums = daw_track_add("drums")
  bind clip = daw_clip_add_midi(drums, 0, 4)
  bind steps = [0, 1, 2, 3]
  bind i = 0
  while i < 4 {
    daw_note_add(drums, clip, i, 0.25, 36, 0.9)
    daw_note_add(drums, clip, i + 0.5, 0.25, 42, 0.6)
    bind i = i + 1
  }
  daw_bounce("pattern.wav", 0, 4, 0.0, 16, "wav", 0.5)
  bind t = music_load("pattern.wav")
  println("seconds ", round(music_duration(t)))
  println("tempo ", round(music_bpm(t)))
  println("onsets ", len(music_onsets(t)))
}
```

```
OUTPUT
seconds 2
tempo 120
onsets 7
```

`daw_note_add(track, clip, beat, length, key, velocity)` places notes relative to the clip. Kick is key 36 and closed hat 42. Under the default JIT, errors in `daw_*` calls are swallowed, so run once with `--interp` while developing.

Export a tune to MIDI and read it back

Write a standard MIDI file and check the notes in it.

```
bind start = do {
  daw_new()
  daw_set_bpm(90)
  bind t = daw_track_add("synth")
  bind c = daw_clip_add_midi(t, 0, 4)
  daw_note_add(t, c, 0, 1, 60, 0.8)
  daw_note_add(t, c, 1, 1, 64, 0.8)
  daw_note_add(t, c, 2, 2, 67, 0.8)
  daw_midi_export_project("tune.mid")
  bind m = music_midi_load("tune.mid")
  println(music_midi_count(m), " notes")
  println(music_midi_notes(m))
  println(file_exists("tune.mid"))
}
```

```
OUTPUT
3 notes
[0, 60, 0.6666659712791443, 64, 1.3333319425582886, 67]
true
```

At 90 BPM a beat lasts two thirds of a second, so the note times come back as 0, 0.667 and 1.333 seconds. `music_midi_bars` adds durations for drawing note bars.

Estimate tempo from onset times

Cross-check `music_bpm` by averaging gaps between hits.

```
fn bpm_from_onsets(times, hits_per_beat) {
  bind total = 0
  bind i = 1
  while i < len(times) {
    bind total = total + times[i] - times[i - 1]
    bind i = i + 1
  }
  60 / ((total / (len(times) - 1)) * hits_per_beat)
}

bind start = do {
  println(round(bpm_from_onsets([0, 0.5, 1, 1.5, 2], 1)))
  println(round(bpm_from_onsets([0, 0.25, 0.5, 0.75, 1], 2)))
}
```

```
OUTPUT
120
120
```

If hits come once per beat, BPM is 60 over the average gap; if there are two hits per beat, multiply the gap by the number of hits first.

Beat clock for visuals

Know which beat you are on and how far through it, from the clock and the tempo.

```
fn beat_number(seconds, bpm) { floor(seconds * bpm / 60) }
fn beat_phase(seconds, bpm) { fract(seconds * bpm / 60) }

bind start = do {
  for t in [0, 0.25, 0.5, 1.2, 3] {
    println(t, "s at 120 BPM: beat ", beat_number(t, 120), ", phase ",
round(beat_phase(t, 120) * 100) / 100)
  }
}
```

```
OUTPUT
0s at 120 BPM: beat 0, phase 0
0.25s at 120 BPM: beat 0, phase 0.5
0.5s at 120 BPM: beat 1, phase 0
1.2s at 120 BPM: beat 2, phase 0.4
3s at 120 BPM: beat 6, phase 0
```

Pulse a sprite with `1 - beat_phase(...)` (big at the start of each beat, shrinking) to sync visuals to music, using the same clock the audio uses.

Rhythm timing grades

Judge how close a key press was to the beat.

```
fn grade_for(press, beat_time) {
  bind delta = abs(press - beat_time) * 1000
  bind g = music_judge(delta)
  music_grade_name(g)
}

bind start = do {
  println(grade_for(1.005, 1))
  println(grade_for(1.045, 1))
  println(grade_for(0.92, 1))
  println(grade_for(1.4, 1))
}
```

```
OUTPUT
PERFECT
GREAT
GOOD
MISS
```

`music_judge(delta_ms)` maps a timing error in milliseconds to a grade index; `music_grade_name(index)` names it (PERFECT, GREAT, GOOD, OK, MISS). Keep the beat times in a list and compare each press with the nearest.

Beep on a button

Play a short synthesised blip with no sound files.

```
GUI
bind start = do {
  open_window(320, 160, "Blip")
  while window_is_open() {
    fill(10, 14, 28)
    if ui_button(100, 60, 120, 40) == 1 { audio_blip(880, 0.08, "square", 0.2) }
    if key_pressed("space") { ui_sound("confirm") }
    present()
  }
}
```

`audio_blip(freq, duration, wave, amp)` takes the waves `sine`, `square`, `saw`, `tri` and `noise`; `ui_sound(name)` plays the built-ins `click`, `hover`, `confirm`, `error`, `toggle` and `tick`. Both are fire-and-forget. Listen for the result: no output can show you the sound.

A positional sound effect

Play a one-shot at a place in the world so it pans with the listener.

```
GUI
bind start = do {
  open_window(320, 160, "Positional")
  bind next_hit = 0.0
  while window_is_open() {
    bind now = time_now()
    bind x = 4 * sin(now)
    if now > next_hit {
      audio_sfx(x, 0, 6, 1, 220, 0.3, 0.12, "saw")
      bind next_hit = now + 0.6
    }
    fill(10, 14, 28)
    set_color(255, 200, 80)
    draw_disc(160 + x * 30, 80, 8)
    present()
  }
}
```

`audio_sfx(x, y, z, w, freq, amp, duration, wave)` places the effect at x, y, z (w is a fourth-dimension coordinate; use 1). The listener sits at the camera. The dot on screen follows the same x so you can see where the sound should come from.

A drone with a tremolo

Start a continuous tone and retune it.

```
GUI
bind start = do {
  open_window(320, 160, "Drone")
  while window_is_open() {
    bind t = time_now()
    audio_tone(0, 0, 0, 6, 1, 110 + 4 * sin(t * 0.5), 0.1, 4, 0.02)
    fill(10, 14, 28)
    set_color(120, 255, 160)
    ui_text(20, 20, 14, "DRONE 110 HZ")
    present()
  }
}
```

`audio_tone(slot, x, y, z, w, freq, amp, lfo_rate, lfo_depth)` creates or retunes the oscillator in a numbered slot. Calling it every frame with a slowly changing frequency glides the pitch; the last two arguments add a built-in wobble.

Axis-aligned box collision

The test behind most 2D games: do two rectangles overlap?

```
form Box { x, y, w, h, }

fn overlaps(a, b) {
  a.x < b.x + b.w && a.x + a.w > b.x && a.y < b.y + b.h && a.y + a.h > b.y
}

bind start = do {
  bind player = Box(10, 10, 20, 20)
  println(overlaps(player, Box(25, 25, 20, 20)))
  println(overlaps(player, Box(30, 10, 20, 20)))
  println(overlaps(player, Box(100, 100, 5, 5)))
}
```

```
OUTPUT
true
false
false
```

Two boxes overlap unless one is entirely to the left, right, above or below the other. Touching edges (30 against 30 above) count as not overlapping.

Circle collision and push-out

Detect two circles touching and compute how far to push them apart.

```
fn gap(ax, ay, ar, bx, by, br) {
  bind dx = bx - ax
  bind dy = by - ay
  sqrt(dx * dx + dy * dy) - (ar + br)
}

fn push_out(ax, ay, ar, bx, by, br) {
  bind dx = bx - ax
  bind dy = by - ay
  bind d = max(sqrt(dx * dx + dy * dy), 0.0001)
  bind overlap = (ar + br) - d
  if overlap <= 0 { [0, 0] } else {
    bind fix = [dx / d * overlap, dy / d * overlap]
    fix
  }
}

bind start = do {
  println(gap(0, 0, 10, 25, 0, 10))
  println(gap(0, 0, 10, 15, 0, 10))
  println(push_out(0, 0, 10, 15, 0, 10))
}
```

```
OUTPUT
5
-5
[5, 0]
```

A negative gap means overlap. Move the second circle by `push_out` and the pair just touches. Splitting the move half-and-half between both circles looks fairer in a crowd.

Projectile arc preview

Predict where a thrown object will be, for aiming guides.

```
fn arc_point(x0, y0, speed, angle_deg, t, g) {
  bind a = angle_deg * 3.14159265 / 180
  bind pt = [x0 + speed * cos(a) * t, y0 + speed * sin(a) * t - 0.5 * g * t * t]
  pt
}

fn flight_time(y0, speed, angle_deg, g) {
  bind a = angle_deg * 3.14159265 / 180
  bind vy = speed * sin(a)
  bind total = (vy + sqrt(vy * vy + 2 * g * y0)) / g
  total
}

bind start = do {
  bind T = flight_time(0, 20, 45, 9.81)
  println("flight time ", round(T * 100) / 100, " s")
  println("landing x ", round(arc_point(0, 0, 20, 45, T, 9.81)[0] * 10) / 10)
}
```

```
OUTPUT
flight time 2.88 s
landing x 40.8
```

(`bind total = ...` before the result is deliberate: a line that begins with `(` or `[` straight after another statement is read as a call or an index on that statement.) This is a y-up formula, like the `rb_*` world. For screen drawing, subtract y from the screen ground line. Plot `arc_point` at ten or twelve values of `t` as small discs for a dotted aiming arc.

Dropping a body in the physics world

Add a rigid body, step the world, read the position.

```
bind start = do {
  rb_gravity(0, -9.81, 0)
  bind box = rb_add(0, 4, 0, 1)
  bind f = 0
  while f < 200 {
    rb_step(0.016)
    rb_floor(box, 0, 0.3, 0.8)
    bind f = f + 1
  }
  bind p = rb_pos(box)
  println("rests at y = ", round(p[1] * 100) / 100)
}
```

```
OUTPUT
rests at y = 0.5
```

Physics y is up: gravity is negative and the floor is at 0, so a resting unit cube has its centre at 0.5. Call `rb_floor` for every body every frame.

A soft ball bounce

The deformable-body world (y down, like the screen).

```
bind start = do {
  bind ball = soft_ball(0, 0 - 3, 0, 1)
  bind lowest = 0 - 99
  bind f = 0
  while f < 90 {
    soft_step(ball, 0.016, 9.8)
    soft_bounce(ball, 2, 0.5)
    bind lowest = max(lowest, soft_centroid(ball)[1])
    bind f = f + 1
  }
  println("hit the floor: ", lowest > 0.9)
  println("nodes ", len(soft_nodes(ball)) / 3)
}
```

```
OUTPUT
hit the floor: true
nodes 86
```

`soft_*` and `rb_*` are different worlds with opposite y directions. Never mix their coordinates without converting.

Invulnerability frames

After a hit, ignore damage for a short time so one collision cannot kill in a single frame.

```
form Hero { hp, safe_until, }

fn take_hit(h, damage, now) {
  if now < h.safe_until { h } else { Hero(max(0, h.hp - damage), now + 1.0) }
}

bind start = do {
  bind h = Hero(10, 0)
  bind h = take_hit(h, 3, 0.1)
  println(h.hp)
  bind h = take_hit(h, 3, 0.5)
  println(h.hp, "  (still protected)")
  bind h = take_hit(h, 3, 1.2)
  println(h.hp)
}
```

```
OUTPUT
7
7  (still protected)
4
```

Store *when* protection ends rather than a countdown, then there is nothing to decrement each frame. Flash the sprite while `now < safe_until`.

Score multiplier with a combo timer

Reward quick chains: each action within a time window raises the multiplier.

```
form Combo { count, last, score, }

fn hit(c, points, now) {
  bind chain = if now - c.last <= 1.5 { c.count + 1 } else { 1 }
  Combo(chain, now, c.score + points * min(chain, 5))
}

bind start = do {
  bind c = Combo(0, 0 - 99, 0)
  bind times = [0, 0.5, 1.0, 1.4, 4.0]
  bind i = 0
  while i < len(times) {
    bind c = hit(c, 100, times[i])
    println("t=", times[i], " chain ", c.count, " score ", c.score)
    bind i = i + 1
  }
}
```

```
OUTPUT
t=0 chain 1 score 100
t=0.5 chain 2 score 300
t=1 chain 3 score 600
t=1.4 chain 4 score 1000
t=4 chain 1 score 1100
```

The loop is a `while` because a `for` body cannot rebind `c` for the next pass. The chain resets when more than 1.5 seconds pass between hits.

High score file

Remember the best score between runs.

```
fn load_best() {
  if file_exists("best.txt") { to_number(trim(read_file("best.txt"))) } else { 0 }
}

fn save_if_better(score) {
  if score > load_best() {
    write_file("best.txt", str(score))
    true
  } else {
    false
  }
}

bind start = do {
  remove_file("best.txt")
  println(save_if_better(120))
  println(save_if_better(80))
  println(save_if_better(300))
  println(load_best())
  remove_file("best.txt")
}
```

```
OUTPUT
true
false
true
300
```

A single-number text file is the simplest save format. Relative paths resolve from the folder you launched the program in, so use a fixed location for a shipped game.

A game phase machine

Menu, playing, paused and game over as a `choose`, with one function for the transitions.

```
choose Phase { Menu, Playing(score), Paused(score), Over(score) }

fn next_phase(p, pressed) {
  match p {
    Menu() => if pressed == "start" { Playing(0) } else { p },
    Playing(s) => if pressed == "pause" { Paused(s) } else if pressed == "die" {
Over(s) } else { Playing(s + 10) },
    Paused(s) => if pressed == "pause" { Playing(s) } else { p },
    Over(s) => if pressed == "start" { Playing(0) } else { p },
    _ => Menu()
  }
}

bind start = do {
  bind p = Menu()
  bind keys = ["junk", "start", "tick", "tick", "pause", "tick", "pause", "tick",
"die", "start"]
  bind i = 0
  while i < len(keys) {
    bind p = next_phase(p, keys[i])
    println(keys[i], " -> ", p)
    bind i = i + 1
  }
}
```

```
OUTPUT
junk -> Menu
start -> Playing(0)
tick -> Playing(10)
tick -> Playing(20)
pause -> Paused(20)
tick -> Paused(20)
pause -> Playing(20)
tick -> Playing(30)
die -> Over(30)
start -> Playing(0)
```

Each phase carries the data it needs (the score). The `Paused` phase ignores everything but `pause`, so nothing advances. Draw functions `match` on the same variants.

Wave spawner with a seeded generator

Produce the same enemy waves for a given seed, so players can share a run.

```
fn next_seed(s) { (s * 75 + 74) % 65537 }

fn wave(seed, level) {
  bind out = list_new()
  bind s = seed + level * 31
  bind n = 3 + level * 2
  bind i = 0
  while i < n {
    bind s = next_seed(s)
    bind kind = if s % 5 == 0 { "tank" } else if s % 2 == 0 { "runner" } else { "grunt"
}
    bind out = list_push(out, kind)
    bind i = i + 1
  }
  out
}

bind start = do {
  println(wave(7, 1))
  println(wave(7, 1))
  println(wave(7, 2))
  println(wave(8, 1))
}
```

```
OUTPUT
[runner, grunt, grunt, runner, grunt]
[runner, grunt, grunt, runner, grunt]
[grunt, grunt, grunt, runner, grunt, runner, grunt]
[grunt, runner, tank, runner, grunt]
```

Equal seed and level always give the same wave. A game that uses `rand()` instead cannot be replayed.

Wrap an entity around the screen

Leave one edge, come back on the opposite one, as in Asteroids.

```
fn wrap(v, size) { ((v % size) + size) % size }

bind start = do {
  println(wrap(650, 640), " ", wrap(0 - 10, 640), " ", wrap(320, 640))
}
```

```
OUTPUT
10 630 320
```

The double modulo handles negative values, since `%` of a negative number is negative.

Unified keyboard and gamepad input

One movement vector, whatever the player is holding.

```
GUI
fn axis(neg, pos) {
  bind v = 0
  if neg { bind v = v - 1 }
  if pos { bind v = v + 1 }
  v
}

fn deadzone(v) { if abs(v) < 0.2 { 0 } else { v } }

bind start = do {
  open_window(320, 200, "Input")
  bind x = 160
  bind y = 100
  bind last = time_now()
  while window_is_open() {
    bind now = time_now()
    bind dt = min(now - last, 0.05)
    bind last = now
    pad_poll()
    bind dx = axis(key_down("left") || key_down("a"), key_down("right") ||
key_down("d")) + deadzone(pad_lx(0))
    bind dy = axis(key_down("up") || key_down("w"), key_down("down") || key_down("s"))
+ deadzone(pad_ly(0))
    bind x = clamp(x + clamp(dx, -1, 1) * 150 * dt, 10, 310)
    bind y = clamp(y + clamp(dy, -1, 1) * 150 * dt, 10, 190)
    fill(10, 14, 28)
    set_color(255, 200, 80)
    draw_disc(x, y, 10)
    present()
  }
}
```

Both devices feed the same `dx, dy`, clamped to -1..1 so holding a key and pushing a stick does not double the speed. With no pad attached `pad_poll()` simply returns 0 and the axes read 0. (Keys and sticks need a person to test.)

Explosion with particles

A burst of sparks at a point, using the particle system with the 2D-to-3D pixel mapping.

```
GUI
fn rgb(r, g, b) { r * 65536 + g * 256 + b }

bind start = do {
  open_window(480, 270, "Burst")
  bind boom_at = 0.0
  bind last = time_now()
  while window_is_open() {
    bind now = time_now()
    bind dt = min(now - last, 0.05)
    bind last = now
    if now >= boom_at {
      pfx_burst((300 - 240) / 40, (120 - 135) / 40, 4, 60, 5, 1.2, 0.3, 0, rgb(255,
220, 90), rgb(255, 50, 10), 0, 1, 0, 0, 0)
      bind boom_at = now + 2
    }
    pfx_step(dt, 4, 0.5, 1000, 0.4)
    fill(10, 14, 28)
    pfx_draw()
    present()
  }
}
```



Figure: a `pfx_burst` a moment after it fires.

With the default camera, putting particles at depth `z = 4` makes one world unit equal `windowheight / 9` pixels (30 pixels in this 270-pixel-high window), so a pixel position `(px, py)` is world `((px - width/2) / 30, (py - height/2) / 30, 4)`. Fixed seeding means the burst looks the same every run.

CHAPTER 14

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

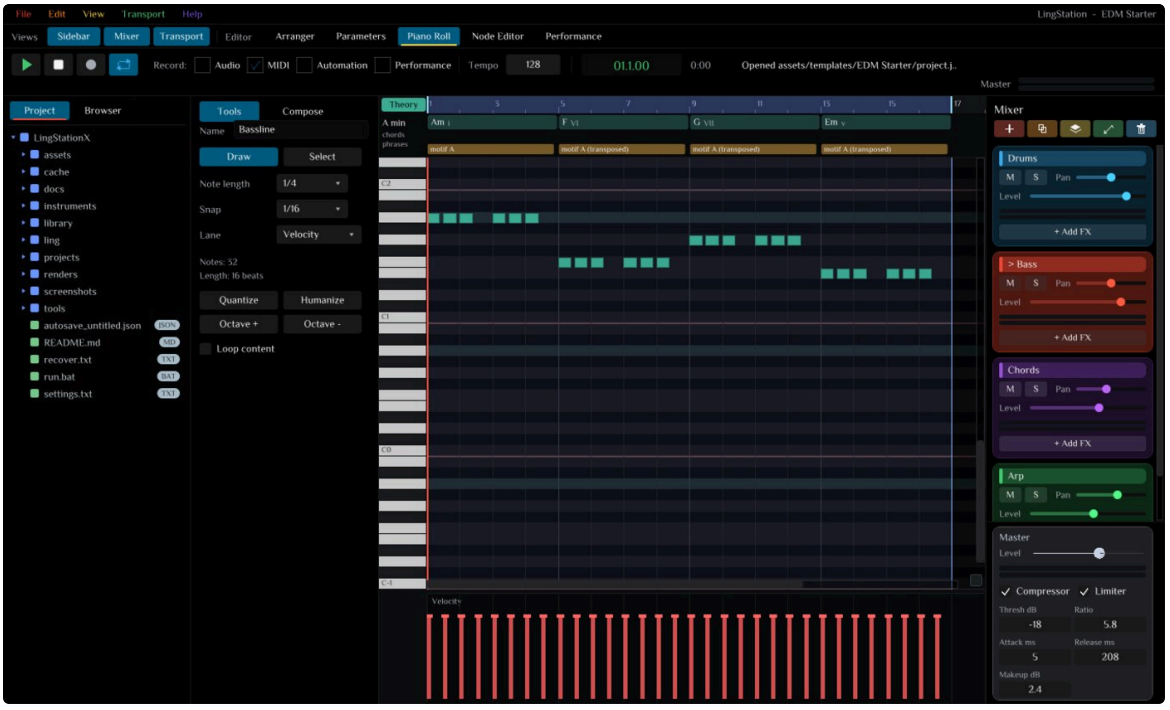


Figure 10.1. The piano roll. The grid, the notes, the keyboard strip and the velocity bars below are all drawn by `roll.ling` with `px_*` calls.

- **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").

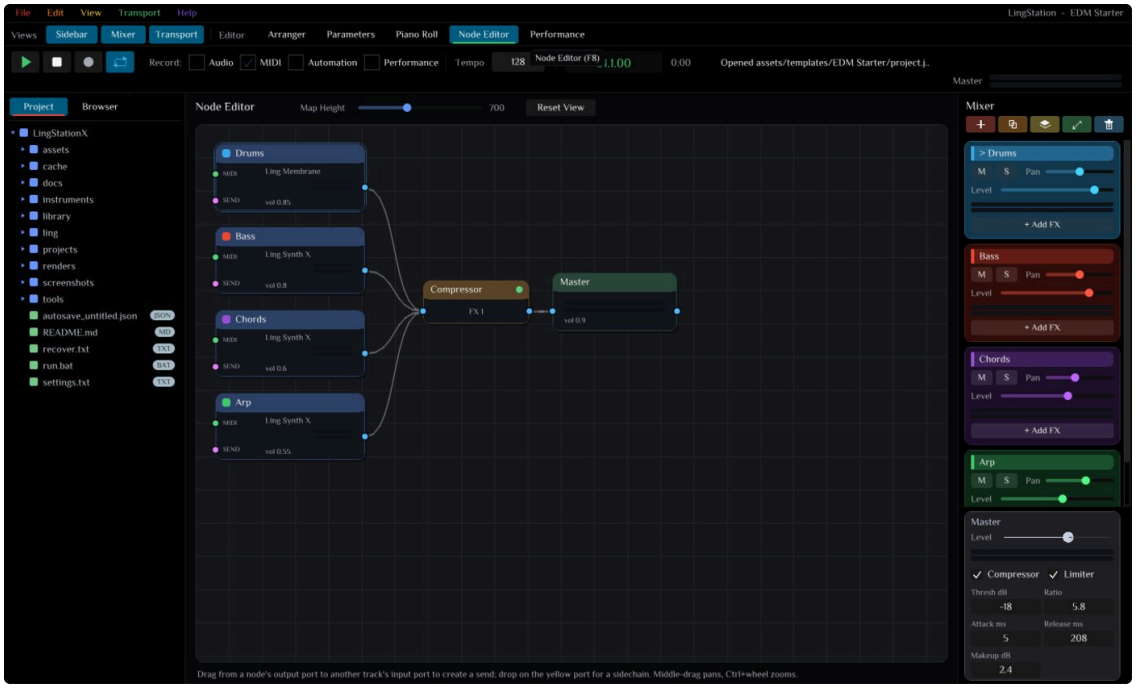


Figure 10.2. The node editor, a second view of the same project: each track is a node, and a cable is a send or a sidechain feeding another track.

- **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 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`.

```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

Reference

Keywords and builtins through the intermediate level.

CHAPTER 16

Keywords

Ling source can be written in many human languages. The lexer (`src/lexer/mod.rs`) maps each spelling of a keyword to one canonical token, so the same program can mix scripts. This chapter lists every keyword with its English spelling, the spellings accepted in Chinese (zh), Japanese (ja), Korean (ko) and Thai (th), what it really does, and a verified example. The lexer also accepts spellings in French, German, Spanish, Portuguese, Russian, Arabic, Persian, Hebrew, Urdu and Hindi, which are not listed individually.

Each keyword carries a **status**:

- **Active** - parsed and given meaning by the language.
- **Parsed, no effect** - accepted by the parser but ignored at run time.
- **Reserved word only** - recognised by the lexer so it cannot be written in another language by accident, but the parser treats it as a plain identifier; it does nothing. Several of these (`stop`, `again`, `change`, `sure`, ...) appear in older documentation as if they worked. They do not.

> Where this chapter and the repository documentation (`docs/src/glossary/keywords.md`, `docs/src/multilingual/keywords.md`) disagree, this chapter follows what the lexer and runtime actually do. Notably the lexer, not the lexicon `.ling` files, is authoritative: for example Thai `choose` is `เลือกแบบ` (not `เลือก`), Thai `form` is `โครงสร้าง` and Japanese `choose` is `選択`.

Examples marked `ling` were run with the default `ling run` backend; where `ling run --interp` prints something different, both outputs are shown. `ling-interp` examples only work on the interpreter.

Keywords can be used as variable names (the parser accepts any keyword wherever a name is expected), which is why the reserved-word-only entries cannot misbehave in old code.

Keyword summary

Keyword	zh	ja	ko	th	Status
`bind`	`令` `灵符`	`束縛` `バ`	`바인드` `묶`	`ผูก`	Active

Keyword	zh	ja	ko	th	Status
do	执	執 実行	실행	ทำ	Active
fn	函	関 関数	함수	ฟังก์ชัน	Active
mod	核 模	モジュール	모듈	โมดูล	Active (functions only)
type	符	型	타입	ชนิด	Parsed, no effect
use	载 引	使う	사용	ใช้ นำเข้า	Active
if	若 如	もし	만약 조건	ถ้า	Active
else	否则 否	他	아니면	มิฉะนั้น	Active
while	循 当 一方	間	동안 반복	ขณะที่	Active
for	历	繰 ために	위해	สำหรับ	Active
in	于	の中	안에	ใน	Active
match	配	一致	매치	จับคู่	Active
return	归	戻る 帰る	반환 귀환	คืน	Active
own	拥有 独	所有	소유	เป็นเจ้าของ	Parsed, no effect
lend	借	貸す	빌려	ให้ยืม	Parsed, no effect
share	共享 共	共有	공유	แบ่งปัน	Parsed, no effect
move	移动 移	移動	이동	ย้าย	Parsed, no effect
copy	复制 复	複製	복사	คัดลอก	Parsed, no effect
async	异步 异	非同期	비동기	ไม่พร้อมกัน	Parsed, no effect
wait	等待 待	待つ	기다려	รอ	Unreliable
asm	-	-	-	-	Parsed, no effect
post	发布 出	投稿	게시	ส่ง	Reserved word only
give	给 予	渡す	전달	ให้	Reserved word only
fit	适合	適合	적합	เหมาะสม	Reserved word only
form	形式 形	構造	구조	โครงสร้าง	Active
choose	选择 选	選択	선택	เลือกแบบ	Active
can	能	できる	가능	สามารถ	Reserved word only

Keyword	zh	ja	ko	th	Status
`change`	改变 变	変える	변경	เปลี่ยนแปลง	Reserved word only
`stop`	止	停止 止まれ 停め	멈춤	หยุด	Reserved word only (no break)
`again`	继续	継続	계속	ทำอีก	Reserved word only (no continue)
`try`	尝试 试	試す	시도	ลอง	Active as a builtin function
`sure`	确定 确	確か	확실	แน่นอน	Reserved word only
`maybe`	可能 或	多分	아마도	อาจจะ	Reserved word only
`pure`	纯	純粋	순수	บริสุทธิ์	Reserved word only
`spawn`	生成 启	起動	생성	สร้าง	Reserved word only
`ok`	好 可	良い	좋아	ตกลง	Active
`bad`	坏 误	悪い	나쁨	ผิดพลาด	Active
`none`	无	なし	없음	ไม่มี	Unreliable
`as`	为	として	로서	เป็น	Active (aliasing)
`where`	条件	但し	단	โดยที่	Skipped
`true` / `false`	真 / 假	真 / 偽	참 / 거짓	จริง / เท็จ	Active (literals)

Keyword reference

`bind`

Introduces a binding: `bind name = expression`. It is the only way to create or change a variable (there is no separate assignment statement), so re-binding an existing name is how you "update" it.

- **Status:** Active
- **Aliases:** zh: `令`, `灵符`; ja: `束縛`, `バ`; ko: `바인드`, `묶`; th: `ผูก` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** At top level, `bind` also defines global values and the program entry point (`bind start = do { ... }`).
- **Note:** Any keyword may be used as a binding name, e.g. `bind change = 5`.

```
bind start = do {
  bind price = 40
  bind price = price + 2
  println(price)
}
```

```
OUTPUT
42
```

`do`

Block expression: `do { statements }` runs its statements in order and evaluates to the value of the last one. A bare `{ ... }` in expression position is the same thing. The program entry point is a `do` block bound to `start`.

- **Status:** Active
- **Aliases:** zh: `执`; ja: `執`, `実行`; ko: `실행`; th: `ทำ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** Under the interpreter, bindings made inside a `do` block are discarded when the block ends. The default JIT backend lets them leak into the enclosing scope. Do not depend on either.

```
bind start = do {
  bind total = do {
    bind a = 20
    bind b = 22
    a + b
  }
  println(total)
}
```

```
OUTPUT
42
```

`fn`

Declares a named function: `fn name(parameters) { body }`. The value of the last expression in the body is the result; `return` exits early. Functions are declared at top level (or inside a `mod`), not inside other functions.

- **Status:** Active
- **Aliases:** zh: `函`; ja: `関`, `関数`; ko: `함수`; th: `ฟังก์ชัน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** Parameter and return type annotations (`a: number`, `-> number`) are parsed and ignored. Generic parameters (`<T>`) are skipped.
- **Note:** A function sees top-level `bind` values read-only, but not the local variables of its caller.
- **Note:** `fn` before a `|...|` is not how closures are written; use `|x| expression` (see **Closures** in the syntax chapter).

```
fn area(w, h) { w * h }
fn describe(n) {
  if n > 100 { return "big" }
  "small"
}
bind start = do {
  println(area(6, 7))
  println(describe(500))
  println(describe(5))
}
```

```
OUTPUT
42
big
small
```

`mod`

Declares a namespace: `mod name { items }`. Functions inside are called as `name::function(...)`.

- **Status:** Active (functions only)

- **Aliases:** zh: `核`, `模`; ja: `モジュール`; ko: `모듈`; th: `โมดูล` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** Values bound inside a `mod` are not usable: `geo::unit` evaluates to `()` under the JIT backend and to the text `geo::unit` under the interpreter, and even the functions of the module itself cannot see them (`undefined` on the interpreter, `()` on the JIT). Put constants in functions instead.

```
mod geo {
  fn area(w, h) { w * h }
  fn perimeter(w, h) { 2 * (w + h) }
}
bind start = do {
  println(geo::area(3, 4))
  println(geo::perimeter(3, 4))
}
```

```
OUTPUT
12
14
```

`use`

Imports another source file: `use "path"` or `use "path" as alias`. With an alias, its functions are called `alias::name(...)`; without one they are called by plain name.

- **Status:** Active
- **Aliases:** zh: `载`, `引`; ja: `使う`; ko: `사용`; th: `ใช้`, `นำเข้า` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** The path is a string, resolved relative to the importing file; the `.ling` extension may be omitted.
- **Note:** Imported top-level `bind` values are not visible to the imported functions under the interpreter and read as `()` under the JIT backend. Share data through functions.

```
use "mathutil.ling" as mu
bind start = do {
  println(mu::cube(3))
}
```

```
OUTPUT
27
```

`if`

Conditional expression: `if condition { ... }`, optionally followed by `else if ... { }` and `else { }`. It evaluates to the value of the branch taken (`()` when no branch is taken and there is no `else`). Braces are mandatory; the condition needs no parentheses.

- **Status:** Active
- **Aliases:** zh: `若`, `如`; ja: `もし`; ko: `만약`, `조건`; th: `ถ้า` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** Use real booleans in conditions. Truthiness of non-booleans differs between backends: the JIT treats `""` and `[]` as true, the interpreter as false.
- **Note:** Bindings made inside an `if` body remain visible after it (the body shares the enclosing scope).

```
bind start = do {
  bind n = 15
  bind kind = if n % 15 == 0 { "fizzbuzz" } else if n % 3 == 0 { "fizz" } else {
"other" }
  println(kind)
}
```

```
OUTPUT
fizzbuzz
```

`else`

Introduces the alternative branch of an `if`, or chains with another `if` as `else if`.

- **Status:** Active
- **Aliases:** zh: `否则`, `否`; ja: `他`; ko: `아니면`; th: `มิฉะนั้น` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)

```
bind start = do {
  bind t = 3
  if t > 5 { println("hot") } else { println("mild") }
}
```

```
OUTPUT
mild
```

`while`

Loop while a condition holds: `while condition { body }`. The body runs in the **enclosing** scope, so `bind counter = counter + 1` inside the loop updates the outer `counter`. `while` is therefore the loop to use when you need to accumulate a result.

- **Status:** Active
- **Aliases:** zh: `循`, `当`, `一方`; ja: `間`; ko: `동안`, `반복`; th: `ขณะที่` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** There is no working `break` or `continue` (see `stop` and `again`). To leave early, make the condition false or `return` from a function.

```
bind start = do {
  bind i = 1
  bind product = 1
  while i <= 5 {
    bind product = product * i
    bind i = i + 1
  }
  println(product)
}
```

```
OUTPUT
120
```

`for`

Iterates over a range, list or string: `for name in iterable { body }`. Ranges are written `lo..hi` and exclude `hi`. Strings iterate by character under the interpreter.

- **Status:** Active
- **Aliases:** zh: `历` ; ja: `繰` , `ために` ; ko: `위해` ; th: `สำหรับ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note: Each iteration gets a fresh copy of the scope**: `bind total = total + i` inside a `for` body does not change the outer `total`. Use `while` to accumulate, or build results with `list_push` on the JIT backend.
- **Note:** Iterating a string yields empty values under the default JIT backend; iterate a list or a range instead.

```
bind start = do {
  for i in 1..4 { println(i * i) }
  for name in ["ada", "lin"] { println(str_upper(name)) }
}
```

```
OUTPUT
1
4
9
ADA
LIN
```

`in`

Separates the loop variable from the sequence in a `for` statement.

- **Status:** Active
- **Aliases:** zh: `于` ; ja: `の中` ; ko: `안에` ; th: `ใน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  for x in [10, 20, 30] { println(x) }
}
```

```
OUTPUT
10
20
30
```

`match`

Pattern matching: `match value { pattern => expression, ... }`. The first arm whose pattern fits is evaluated. Patterns: `_` (anything), a literal (number, string, `true` / `false`), a bare name (matches anything and binds it), `ok(p)` / `bad(p)`, and enum variants `Variant(p1, p2)` (a payload-less variant is written `Variant()`).

- **Status:** Active
- **Aliases:** zh: `配`; ja: `一致`; ko: `매치`; th: `จับคู่` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** If no arm matches, the whole `match` evaluates to `()`; there is no exhaustiveness check.
- **Note:** There are no guards (`if` conditions) and no patterns for lists, ranges or `form` records.

```
choose Shape { Circle(r), Rect(w, h), Dot }
fn area(s) {
  match s {
    Circle(r) => 3 * r * r,
    Rect(w, h) => w * h,
    Dot() => 0
  }
}
bind start = do {
  println(area(Circle(2)))
  println(area(Rect(3, 4)))
  println(area(Dot()))
  println(match 7 { 1 => "one", 7 => "seven", _ => "other" })
}
```

```
OUTPUT
12
12
0
seven
```

`return`

Leaves the current function immediately with a value: `return expression` (or bare `return` for `()`).

- **Status:** Active
- **Aliases:** zh: `归`; ja: `戻る`, `帰る`; ko: `반환`, `귀환`; th: `คืน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** A function without `return` yields its last expression. A `return` directly inside `start` ends the program silently on the JIT backend and is a runtime error (`unexpected top-level return`) on the interpreter.

```
fn first_even(xs) {
  for x in xs {
    if x % 2 == 0 { return x }
  }
  return 0 - 1
}
bind start = do {
  println(first_even([3, 5, 8, 9]))
  println(first_even([1, 3]))
}
```

```
OUTPUT
8
-1
```

`own`

Marks a value as owned: `own expression`. These five words are prefix operators that the parser accepts and then ignores: the value is evaluated unchanged and **no ownership or borrow checking happens** at run time.

- **Status:** Parsed, no effect
- **Aliases:** zh: `拥有`, `独`; ja: `所有`; ko: `소유`; th: `เป็นเจ้าของ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** In function parameter lists, a leading `own`, `lend` or `&` before a parameter name is skipped.
- **Note:** Use-after-move is not detected: the example below runs normally.

```
bind start = do {
  bind a = own [1, 2, 3]
  bind b = move a
  println(a)
  println(b)
}
```

```
OUTPUT
[1, 2, 3]
[1, 2, 3]
```

`lend`

Marks a read-only borrow: `lend expression`. These five words are prefix operators that the parser accepts and then ignores: the value is evaluated unchanged and **no ownership or borrow checking happens** at run time.

- **Status:** Parsed, no effect
- **Aliases:** zh: `借`; ja: `貸す`; ko: `빌려`; th: `ให้ยืม` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind a = 5
  bind b = lend a
  println(b)
}
```

```
OUTPUT
5
```

`share`

Marks a shared reference: `share expression`. These five words are prefix operators that the parser accepts and then ignores: the value is evaluated unchanged and **no ownership or borrow checking happens** at run time.

- **Status:** Parsed, no effect
- **Aliases:** zh: `共享`, `共`; ja: `共有`; ko: `공유`; th: `แบ่งปัน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind a = 5
  bind b = share a
  println(b)
}
```

```
OUTPUT
5
```

`move`

Marks a transfer of ownership: `move expression`. These five words are prefix operators that the parser accepts and then ignores: the value is evaluated unchanged and **no ownership or borrow checking happens** at run time.

- **Status:** Parsed, no effect
- **Aliases:** zh: `移动`, `移`; ja: `移動`; ko: `이동`; th: `ย้าย` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind a = 5
  bind b = move a
  println(b)
}
```

```
OUTPUT
5
```

`copy`

Marks an explicit copy: `copy expression`. These five words are prefix operators that the parser accepts and then ignores: the value is evaluated unchanged and **no ownership or borrow checking happens** at run time.

- **Status:** Parsed, no effect
- **Aliases:** zh: `复制`, `复`; ja: `複製`; ko: `복사`; th: `คัดลอก` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind a = 5
  bind b = copy a
  println(b)
}
```

```
OUTPUT
5
```

`form`

Declares a record type: `form Name { field, field }`. Create an instance by calling the name like a function with the fields in order, and read a field with `value.field`.

- **Status:** Active
- **Aliases:** zh: `形式`, `形`; ja: `構造`; ko: `구조`; th: `โครงสร้าง` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

- **Note:** Field types may be written (`x: number`) but are ignored. **A type annotation directly before the closing brace swallows the brace**, so when you annotate fields, end the list with a comma: `form P { x: number, y: number, }`.
- **Note:** Instances are immutable; to "change" a field build a new instance. Two instances compare unequal with `==` even when their fields are equal.
- **Note:** Struct literals such as `Point { x: 1, y: 2 }` are not supported.

```
form Point { x, y }
bind start = do {
  bind p = Point(3, 4)
  println(p)
  println(p.x + p.y)
}
```

```
OUTPUT
Point { x: 3, y: 4 }
7
```

`choose`

Declares an enumeration with optional payloads: `choose Name { Variant, Variant(a, b) }`. Build values by calling a variant like a function and take them apart with `match`.

- **Status:** Active
- **Aliases:** zh: `选择`, `选`; ja: `選択`; ko: `선택`; th: `เลือกแบบ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** A variant without payload is still constructed with parentheses (`Dot()`), and matched as `Dot()`.
- **Note:** The variant name of a value is available as `value.tag`.
- **Note:** As with `form`, a payload type annotation is only safe inside parentheses.

```
choose Light { Red, Yellow, Green }
fn action(l) {
  match l {
    Red() => "stop",
    Yellow() => "slow",
    Green() => "go"
  }
}
bind start = do {
  println(action(Green()))
  println(Red().tag)
}
```

```
OUTPUT
go
Red
```

`change`

Documented as "mutate a binding". It has no effect: mutation is done by re-binding with `bind`. `change` is an ordinary identifier.

- **Status:** Reserved word only
- **Aliases:** zh: `改变`, `变`; ja: `変える`; ko: `변경`; th: `เปลี่ยนแปลง` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** `change x = x + 1` is a syntax error; write `bind x = x + 1`.

```
bind start = do {
  bind x = 1
  bind x = x + 1
  println(x)
}
```

```
OUTPUT
2
```

`stop`

Documented as "break out of the loop". **It does nothing**: the runtime has no break, so `stop` is an undefined name.

- **Status:** Reserved word only (no break)
- **Aliases:** zh: `止` ; ja: `停止` , `止まれ` , `停め` ; ko: `멈춤` ; th: `หยุด` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** Under the interpreter, `stop` inside a loop raises `undefined: 'stop'` ; on the JIT backend it is silently ignored and the loop keeps running. To end a loop early, use `while` with a flag, or `return` from a function.

```
fn find(xs, target) {
  for x in xs {
    if x == target { return true }
  }
  false
}
bind start = do {
  println(find([1, 2, 3], 2))
  println(find([1, 2, 3], 9))
}
```

```
OUTPUT
true
false
```

`again`

Documented as "continue to the next iteration". **It does nothing**, for the same reason as `stop` . Skip an iteration with an ordinary `if` around the body.

- **Status:** Reserved word only (no continue)
- **Aliases:** zh: `继续` ; ja: `継続` ; ko: `계속` ; th: `ทำอีก` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  for i in 0..6 {
    if i % 2 == 1 { print(i) }
  }
}
```

```
OUTPUT
1
3
5
```

`try`

Not special syntax. `try` is the ordinary builtin `try(closure)`, which runs a closure and converts a runtime error into a `bad(...)` result (see the builtins chapter).

- **Status:** Active as a builtin function
- **Aliases:** zh: `尝试`, `试`; ja: `試す`; ko: `시도`; th: `ลอง` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** The closure is written `| | expr` with a space; the lexer reads `||` as the boolean "or" operator.
- **Note:** Works only on the interpreter.

```
INTERPRETER ONLY
bind start = do {
  println(try(| | 6 * 7))
}
```

```
OUTPUT
Ok(42)
```

The default JIT backend prints this instead:

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

`ok`

Constructs a success result (`ok(value)`) and also serves as a `match` pattern.

- **Status:** Active
- **Aliases:** zh: `好`, `可`; ja: `良い`; ko: `좋아`; th: `ตกลง` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
fn parse_age(s) {
  bind n = to_number(s)
  if n > 0 { ok(n) } else { bad("not a positive number") }
}
bind start = do {
  println(parse_age("41"))
  println(match parse_age("41") { ok(n) => n + 1, bad(e) => e })
}
```

```
OUTPUT
Ok(41)
42
```

`bad`

Constructs a failure result (`bad(reason)`) and serves as a `match` pattern. A `bad` value prints as `Err(reason)`.

- **Status:** Active
- **Aliases:** zh: `坏`, `误`; ja: `悪い`; ko: `나쁨`; th: `ผิดพลาด` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
fn parse_age(s) {
  bind n = to_number(s)
  if n > 0 { ok(n) } else { bad("not a positive number") }
}
bind start = do {
  println(parse_age("x"))
  println(match parse_age("x") { ok(n) => n, bad(e) => "error: " + e })
}
```

```
OUTPUT
Err(not a positive number)
error: not a positive number
```

`as`

Gives a name to an imported module: `use "file" as alias`. Also appears in the (practically unusable) `type Name as Type` form.

- **Status:** Active (aliasing)
- **Aliases:** zh: `为`; ja: `として`; ko: `로서`; th: `เป็น` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
use "mathutil.ling" as mu
bind start = do {
  println(mu::cube(2))
}
```

```
OUTPUT
8
```

`true` and `false`

The boolean literals. They are the only values `if` and `while` should be given.

- **Status:** Active
- **Aliases:** zh: `真`, `假`; ja: `真`, `偽`; ko: `참`, `거짓`; th: `จริง`, `เท็จ` (further spellings exist in other languages)

```
bind start = do {
  println(true && !false)
}
```

```
OUTPUT
true
```

```
bind start = do {
  println(false || 1 == 2)
}
```

```
OUTPUT
false
```

Words that look like keywords but are not

- `start` (and `main`, `启`, `เริ่ม`, `시작`, …) is **not** a keyword. It is the conventional *name* of the entry binding; see the syntax chapter. The runtime recognises a list of entry names in many languages (`start`, `main`, `启`, `เริ่ม`, `시작`, `начать`, `inicio`, `commencer`, `anfang`, `शुरू`, `התחל`, `شروع`, `ابدأ`, …) and otherwise runs the first top-level binding whose value is a `do` block.
- `result` appears in the lexicon files and glossary but the lexer does not recognise it; it is an ordinary identifier.
- `loop`, `break`, `continue`, `struct`, `enum`, `impl`, `trait`, `unsafe`, `const`, `static`, `ref`, `mut`, `pub`, `priv` and `let` appear in some lexicon files (for example the Thai one) but are **not keywords**. Ling has no `loop`, no `let` and no `const`; use `while`, `bind`, and a top-level `bind`.
- Type names (`number`, `text`, `bool`, `list`, `map`, `tuple`) in the glossary are documentation words only. Ling is dynamically typed and has no map type.

CHAPTER 17

Syntax, Types and Semantics

This chapter describes the Ling language as it actually behaves. Everything here was checked against the lexer and parser in `src/` and by running programs. Where the README or the glossary promise something the implementation does not do, the difference is called out.

Ling currently has two execution engines. `ling run file.ling` uses the Cranelift JIT backend by default; `ling run --interp file.ling` uses the tree-walking interpreter, which is the semantic reference. They agree on well-behaved programs but differ in several corners, summarised in the last section of this chapter. Examples are run on both engines and any difference is shown.

Program structure

A Ling file is a sequence of **items** at top level: `bind`, `fn`, `mod`, `form`, `choose`, `use`, `type`. Statements such as `println(...)` are **not** allowed at top level; they must live inside a function or the entry point.

The entry point

Execution starts at a top-level binding whose value is a `do` block. The conventional spelling is:

```
OUTPUT
bind start = do {
  ...
}
```

The runtime looks for a binding named `start`, `main`, `启`, `เริ่ม`, `시작`, `начать`, `начало`, `inicio`, `comenzar`, `début`, `commencer`, `démarrer`, `anfang`, `starten`, `início`, `শুরু`, `شروع`, `ابدأ` or `התחל` (checked in that order). If none exists, the first top-level binding whose value is a `do { }` block is used. If there is no such binding the program is a library and `ling run` reports `no entry point`.

`fn main() { ... }`, as shown in some README examples, does **not** work: `main` must be a `bind` of a `do` block. The error is `no entry point – need bind start = do {...}`.

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

```
OUTPUT
Hello, Ling!
```

```
bind main = do {
  println("main works too")
}
```

```
OUTPUT
main works too
```

Comments and whitespace

A comment runs from `//` or `#` to the end of the line. Whitespace, including newlines, separates tokens; statements need no terminator, and an optional `;` may follow any statement. There are no block comments.

```
# a hash comment
// a slash comment
bind start = do {
  println("a") // trailing comment
  println("b"); println("c")
}
```

```
OUTPUT
a
b
c
```

Identifiers

An identifier starts with a letter or underscore (any Unicode letter, including CJK, Thai, Cyrillic, Arabic) and continues with letters, digits and underscores. Identifiers are case-sensitive. Any keyword may be used as an identifier.

```
bind start = do {
  bind 名前 = "Ling"
  bind ชื่อ = 3
  println(名前)
  println(ชื่อ + 1)
}
```

```
OUTPUT
Ling
4
```

Literals

Literal	Form	Notes
Number	`42`, `3.14`	Decimal digits with an optional single fractional part. There is **no** hexadecimal, exponent (`1e3`), digit separator (`1_000`), or leading/trailing-dot (`.5`, `3.`) form. All numbers are 64-bit floats.
Text	`"hello"`	Double quotes only; may span several lines. Escapes: `\n \t \r \" \\ \0`. Any other `\x` is kept literally, backslash included. No interpolation: `"{x}"` is just those characters.
Boolean	`true`, `false`	Also `真` / `假`, `참` / `거짓`, `จริง` / `เท็จ`, ...
Unit	`()`	The "nothing" value; produced by functions with no result. Prints as `()`.
List	`[1, "two", 3.0]`	Heterogeneous, zero-indexed.
Range	`0..5`	Half-open: 0, 1, 2, 3, 4. Used with `for`.
Function value	`|x| x + 1`	Closure; see below.

There is no character literal (`'a'` is a syntax error), no map/dictionary literal and no tuple literal.

```
bind start = do {
  println("line1\nline2")
  println("tab\there")
  println("quote: \"q\" backslash: \\")
  println(1.5 + 1)
  println([1, "two", 3.5, true])
  println(())
}
```

```
OUTPUT
line1
line2
tab	here
quote: "q" backslash: \
2.5
[1, two, 3.5, true]
()
```

Numbers

Because every number is a 64-bit float, integers are exact up to 2^53 and division always produces a float result.

```
bind start = do {
  println(7 / 2)
  println(10 / 5)
  println(7 % 3)
  println(0.1 + 0.2)
  println(1000000 * 1000000)
}
```

```
OUTPUT
3.5
2
1
0.30000000000000004
1000000000000
```

Division by zero is not an error: under the interpreter `1 / 0` is `inf` and `0 / 0` is `NaN`; the JIT backend prints `()` for `1 / 0`.

Types

Ling is dynamically typed. A value is one of:

Type	Examples	Created by
number	`3`, `2.5`	numeric literals, arithmetic
text	`"abc"`	string literals, `+`, string builtins
bool	`true`	literals, comparisons
unit	`()`	functions without a result, `()`
list	`[1, 2]`	list literals, `list_new`, ranges (conceptually)
result	`ok(5)`, `bad("e")`	the `ok` / `bad` constructors
function	`|x| x`	closure expressions
record	`Point { x: 1, y: 2 }` as printed	a `form` name called like a function
variant	`Circle(2)`	a `choose` variant called like a function

There is **no map/dictionary type**, no set, no tuple, no character type and no null: the builtins `list_*` operate on lists, and keys must be emulated with parallel lists. The type names `number`, `text`, `bool`, `list`, `map`, `tuple` in the glossary and lexicon files are documentation only. Type annotations in code (`x: number`, `-> number`) are parsed and **ignored**.

```
fn describe(a: number, b: text) -> text {
  b + to_str(a)
}
bind start = do {
  println(describe(5, "n="))
}
```

```
OUTPUT
n=5
```

Operators

From the tightest to the loosest binding:

Level	Operators	Associativity	Notes
1	`.` `()` `[]` `::` `..`	left	member/method, call, index, path, range
2	`-` (prefix) `!` `&` `wait` `own` `lend` `share` `move` `copy`	prefix	`-x` is `0 - x`; `!x` is `x == false`; `&x` is just `x`; the ownership words are no-ops
3	`*` `/` `%`	left	
4	`+` `-`	left	`+` concatenates text (the other operand is converted to text)
5	`==` `!=` `<` `>` `<=` `>=`	left	`1 < 2 == true` parses as `(1 < 2) == true`
6	`&&`	left	
7	`||` and a single `|`	left	

Notes that matter in practice:

- **Unary minus works** (`-x`, `-3 + 5`, `2 * -x`) but cannot be doubled (`- - x` is a syntax error). Writing `0 - x` is always safe and is what older documentation recommends.
- **The upper bound of a range must be a single name, number or parenthesised expression.** `0..n` and `0..(n + 1)` work; `0..n + 1` is a syntax error, and `0..len(xs)` does not do what it looks like (the call is applied to the range). Write `0..(len(xs))` or bind the length first.
- **`&&` and `||` do not short-circuit** on the interpreter: both operands are always evaluated, so do not rely on `x != 0 && 10 / x > 1` to protect the right-hand side.
- Comparison chains are left-nested: `1 == 1 == 1` is `(1 == 1) == 1`, which is `false`.
- `+` on a number and text converts: `"a" + 1` is `a1` and `1 + "a"` is `1a`. Arithmetic between a number and a boolean is a runtime error on the interpreter.
- `%` on floats works (`7.5 % 2` is `1.5`).
- Text compares with `==`, `!=`; ordering (`<`) on text is not reliable (JIT gives `false`, the interpreter errors). Lists, records and variants are never equal under `==` (`[1, 2] == [1, 2]` is `false`); compare elements or fields individually. Numbers, text and booleans compare by value.

```
bind start = do {
  println(2 + 3 * 4)
  println((2 + 3) * 4)
  println(10 - 3 - 2)
  println(-3 + 5)
  println(2 * -4)
  println(1 < 2 && 2 < 3)
  println(!true || 1 == 1)
  println("n=" + 5)
  println([1, 2] == [1, 2])
}
```

```
OUTPUT
14
20
5
2
-8
true
true
n=5
false
```

Bindings, scope and shadowing

`bind name = expression` creates a binding. There is **no assignment statement** (`x = 5` is a syntax error); to change a value, bind the name again. A later `bind` of the same name replaces the earlier one (shadowing).

```
bind start = do {
  bind x = 1
  bind x = x + 10
  bind x = x * 2
  println(x)
}
```

```
OUTPUT
22
```

Where a re-binding is visible depends on the construct that contains it:

Construct	Bindings made inside it
`if` / `else` body	stay visible afterwards (shared scope)
`while` body	stay visible afterwards, so loops can accumulate
`for` body	**discarded** at the end of each iteration (fresh copy per iteration)
`match` arm	discarded
function body	local to the call
`do { }` block	discarded by the interpreter, **leaked** into the enclosing scope by the JIT

The consequence that surprises most newcomers: a `for` loop cannot accumulate into an outer variable. Use `while`.

```
bind start = do {
  bind total = 0
  for i in 1..5 { bind total = total + i }
  println(total)
  bind t2 = 0
  bind i = 1
  while i < 5 {
    bind t2 = t2 + i
    bind i = i + 1
  }
  println(t2)
}
```

```
OUTPUT
0
10
```

Top-level `bind` values are global constants. Functions can read them, but each function call starts with only its parameters and those globals; the local variables of the caller are not visible.

```
bind RATE = 3
fn scaled(n) { n * RATE }
bind start = do {
  bind RATE = 100
  println(scaled(2))
}
```

```
OUTPUT
6
```

Functions

`fn name(a, b) { ... }`. The result is the last expression, or the value of `return`. Functions may be recursive and may be declared in any order (an `fn` below `start` is visible to it). Functions cannot be nested inside other functions. Calling a function with too few arguments leaves the missing parameters undefined; using one is a runtime error on the interpreter and yields `()` on the JIT.

```
fn fib(n) {
  if n < 2 { return n }
  fib(n - 1) + fib(n - 2)
}
fn greet(name) { "Hello, " + name + "!" }
bind start = do {
  println(fib(15))
  println(greet("Ling"))
}
```

```
OUTPUT
610
Hello, Ling!
```

A function that is called with an unknown name fails with `unknown function` on the interpreter, but the JIT backend quietly evaluates the call to `()`. Check spelling carefully; use `ling run --interp` to get a diagnostic.

Control flow

if / else

`if` is an expression. Braces are required. See the `if` keyword entry for truthiness caveats.

```
bind start = do {
  bind n = 7
  bind label = if n < 0 { "negative" } else if n == 0 { "zero" } else { "positive" }
  println(label)
}
```

```
OUTPUT
positive
```

while

```
bind start = do {
  bind n = 27
  bind steps = 0
  while n != 1 {
    bind n = if n % 2 == 0 { n / 2 } else { 3 * n + 1 }
    bind steps = steps + 1
  }
  println(steps)
}
```

```
OUTPUT
111
```

for and early exit

Loops over a range, a list (or, on the interpreter, a string). There is no `break` / `continue` (`stop` and `again` are reserved words that do nothing); leave a loop early by `return`ing from a function.

```
fn index_of(xs, target) {
  bind i = 0
  while i < len(xs) {
    if xs[i] == target { return i }
    bind i = i + 1
  }
  0 - 1
}
bind start = do {
  println(index_of(["a", "b", "c"], "b"))
  println(index_of(["a"], "z"))
}
```

```
OUTPUT
1
-1
```

The example uses `while` with an explicit index because a counter updated inside a `for` body would be discarded after each iteration.

---

match

See the `match` keyword entry. Patterns are tried top to bottom.

```
fn classify(n) {
  match n {
    0 => "zero",
    1 => "one",
    other => "many (" + to_str(other) + ")"
  }
}
bind start = do {
  println(classify(0))
  println(classify(1))
  println(classify(9))
}
```

```
OUTPUT
zero
one
many (9)
```

Records and enumerations

`form` defines a record and `choose` a tagged union. Both are used by calling the declared name as a constructor.

```
form Player { name, hp, }
choose Cmd { Move(dx, dy), Say(text), Quit }
fn run(c) {
  match c {
    Move(dx, dy) => "move " + to_str(dx) + "," + to_str(dy),
    Say(t) => "say " + t,
    Quit() => "bye"
  }
}
bind start = do {
  bind p = Player("Lin", 30)
  println(p.name + " has " + to_str(p.hp))
  println(run(Move(1, 0 - 1)))
  println(run(Say("hi")))
  println(run(Quit()))
}
```

```
OUTPUT
Lin has 30
move 1,-1
say hi
bye
```

Constructing with the wrong number of fields is an error on the interpreter (`P expects 2 field(s), got 1`) and silently yields `()` on the JIT backend.

Closures

A closure is written `|params| expression`, for example `|x| x * 2` or `|a, b| a + b`. A closure with no parameters must be written `| | expression` (with a space) because `||` lexes as the logical-or operator. Closures capture the variables visible where they are created.

Closures are the least reliable part of the language. Calling a closure held in a variable (`bind f = |x| x * 2` then `f(3)`) fails on the interpreter (`unknown function 'f'`, because a call by bare name only looks up declared `fn` items) and gives `()` or a correct answer on the JIT depending on the surrounding code. Passing a closure to a function that calls it fails on the JIT. What is dependable:

- Calling a closure expression directly: `(|x| x + 1)(5)`.
- Handing a closure to the builtins that take one (`try`, `list_map`, `list_filter`, `list_find`) **on the interpreter**.

Prefer a named `fn` whenever you need a function value of your own.

INTERPRETER ONLY

```
bind start = do {
  println((|x| x + 1)(5))
  println(list_map([1, 2, 3], |x| x * 10))
}
```

OUTPUT

```
6
[10, 20, 30]
```

The default JIT backend prints this instead:

OUTPUT

```
6
()
```

Modules and imports

Inline modules. `mod name { fn ... }` groups functions under a namespace called with `name::function(...)`.

File imports. `use "path"` loads another file; `use "path" as alias` namespaces it. The path is relative to the importing file and the `.ling` extension may be left off. A file used as a library has no entry point; the entry file is the one passed to `ling run`.

```
mod text_tools {
  fn shout(s) { str_upper(s) + "!" }
}
use "mathutil.ling" as mu
bind start = do {
  println(text_tools::shout("hey"))
  println(mu::cube(4))
}
```

```
OUTPUT
HEY!
64
```

where `mathutil.ling` contains:

```
FRAGMENT
fn cube(n) { n * n * n }
```

Imported files share the limitation noted for `mod`: values bound at the top level of an imported file are not usable; export functions instead.

Ownership words

Ling's design documents describe an ownership system (`own`, `lend`, `share`, `move`, `copy`, `change`). In the shipped runtime these words are accepted as prefix operators on expressions (and before parameter names) and then **ignored**: there is no borrow checker, use-after-move is not detected, and `change` is not even an operator. Memory is managed automatically. They are safe to write as documentation of intent. See the keyword entries.

Concurrency and `async`

`async` and `wait` are accepted syntax with no scheduling behind them: an `async fn` runs synchronously when called, and `wait` returns the value on the interpreter but `()` on the JIT. `spawn` is not implemented. Genuine parallelism is provided by the `thread_*`, `chan_*` and `shared_*` builtins, where each thread is an isolated interpreter that communicates by channels; see the builtins chapter.

Error handling

There are no exceptions and no `try` / `catch` syntax. Recoverable failures are values: a function returns `ok(value)` or `bad(reason)` and callers unpack the result with `match`. Many builtins follow a softer convention instead - they return `0`, `false`, `""` or `[]` on failure (for example `to_number`, `list_get`, `file_exists`). A fatal problem raises a **runtime error** that ends the program with a message and a traceback; `try(closure)` (interpreter only) converts such an error into `bad(message)`.

```
fn parse_pair(text) {
  bind parts = split(text, ",")
  if len(parts) != 2 { return bad("expected two values") }
  ok(to_number(parts[0]) + to_number(parts[1]))
}
bind start = do {
  println(match parse_pair("3,4") { ok(v) => v, bad(e) => e })
  println(match parse_pair("3") { ok(v) => v, bad(e) => e })
}
```

```
OUTPUT
7
expected two values
```

Runtime errors look like this (interpreter):

```
INTERPRETER ONLY
bind start = do {
  println(read_file("no_such_file.txt"))
}
```

```
OUTPUT
error[runtime]: read_file 'no_such_file.txt': The system cannot find the file
specified. (os error 2)
 --> example.ling
  traceback (deepest call last):
    0: start
```

The default JIT backend prints this instead:

```
OUTPUT
()
```

Writing Ling in other languages

Because the lexer maps each keyword spelling to one token, a file may be written entirely in Chinese, Japanese, Korean, Thai and other supported languages, and may mix them. Builtin names have translations too (for example `印` for `print`; see each builtin entry). `ling` prints `[detected language: ...]` on standard error for non-English files. Use `lingfu normalize <lang>` to rewrite a whole project into another language.

```
函 加倍(n) { 归 n * 2 }
令 启 = 执 {
  令 x = 加倍(21)
  若 x > 40 { 印("大") } 否 { 印("小") }
  印(x)
}
```

```
OUTPUT
大
42
[detected language: Chinese (中文)]
```

The same program in Thai:

```
ผูก เริ่ม = ทำ {
  ผูก x = 5
  ถ้า x > 3 { พิมพ์("มาก") } มิฉะนั้น { พิมพ์("น้อย") }
}
```

```
OUTPUT
มาก
[detected language: Thai (ภาษาไทย)]
```

JIT versus interpreter: known differences

These were observed with Ling 2030.1.53. Choose `ling run --interp` when you need diagnostics or the reference behaviour, and the default JIT for speed once a program is correct.

Situation	JIT (default)	Interpreter (`--interp`)
Unknown function name	evaluates to `()` silently	`unknown function` error
Calling a closure stored in a variable	often `()`	`unknown function` error
Passing a closure to a builtin (`try`, `list_map`, ...)	closure becomes `()`	works
`list_push` / `.push` on a list	may mutate the original list	returns a new list
`bind` inside `do { }`	leaks to enclosing scope	discarded
`for c in "text"`	empty values	iterates characters
`wait expr`	`()`	the value
`pi`, `none` as bare names	`()`	`pi` works, `none` undefined
Printing a range value	`[0, 5]` (endpoints)	`[0, 1, 2, 3, 4]`
Index past end of list	yields an empty value	runtime error `index N out of bounds`
`1 / 0`	`()`	`inf`
`for i in 0..len(xs)`	empty loop	`undefined: len`
Truthiness of `""` and `[]`	true	false
String ordering `"a" < "b"`	`false`	runtime error
`read_file` of a missing file	`()`	runtime error

Situation	JIT (default)	Interpreter (`--interp`)
Wrong constructor arity	`()`	runtime error
Top-level `return` in `start`	ends program quietly	runtime error

A program that avoids these corners behaves identically on both engines.

CHAPTER 18

Core Builtins Reference

Every function in this chapter is called by name, for example `sqrt(16)`. Parameters are positional; a missing trailing argument usually takes a documented default instead of raising an error. Each entry lists the spellings that also work in Chinese (zh), Japanese (ja), Korean (ko) and Thai (th); the runtime recognises further aliases in French, German, Russian, Arabic, Persian, Hebrew and Urdu which are not listed here.

Examples marked `ling` were run with the default `ling run` backend and the output shown is real. Where the tree-walking interpreter (`ling run --interp`) prints something different the difference is shown under the example. Examples marked `ling-interp` only work on the interpreter, and `ling-gui` examples were checked against the source but not executed. Blocks marked `ling-module` are library files, not programs.

Graphics, audio, networking, cryptography, physics and AI builtins are documented in their own chapters.

Output and formatting

Console output. `print` and `println` are the same builtin: both write every argument converted to text, **joined with no separator**, followed by a newline. Use `format` or `+` to build spaced output.

Name	Summary
`print(value, ...)`	Writes its arguments to standard output, then a newline
`println(value, ...)`	Identical to `print`: writes the arguments joined without a separator and ends with a newline
`print_color(color, value, ...)`	Prints a line wrapped in a bold bright ANSI colour escape sequence
`print_file(text)`	Writes text to standard output **without** a trailing newline
`format(template, value, ...)`	Builds a string by substituting each `{}` in `template` with the next argument
`format::join(list)`	Concatenates every element of a list as text, with no separator

`print(value, ...)`

Writes its arguments to standard output, then a newline.

- **Aliases:** English: `println`; zh: `印`, `打印`; ja: `印刷`; ko: `출력`; th: `พิมพ์` (further languages exist)
- **Parameters:** Any number of values of any type. Each is converted to text; there is no separator between them.
- **Returns:** `()` (unit).
- **Note:** `print` appends a newline exactly like `println`; there is no newline-free variant (see `print_file` for that).
- **Note:** Lists print as `[1, 2, 3]`, results as `Ok(v)` / `Err(e)`, whole numbers without a decimal point (`3`, not `3.0`).

```
bind start = do {
  print("Hello, Ling!")
  print(42)
  print(3.5)
  print(true)
  print([1, 2, 3])
  print("a", "b", 1 + 2)
}
```

```
OUTPUT
Hello, Ling!
42
3.5
true
[1, 2, 3]
ab3
```

`println(value, ...)`

Identical to `print`: writes the arguments joined without a separator and ends with a newline.

- **Aliases:** English: `print`; zh: `印`, `打印`; ja: `印刷`; ko: `출력`; th: `พิมพ์` (further languages exist)
- **Parameters:** Any number of values.
- **Returns:** `()`.

- **Note:** Gotcha: `println("sum", 1 + 2)` prints `sum3`, not `sum 3`. Put the space inside a string (`"sum "`) or use `format`.

```
bind start = do {
  println("sum", 1 + 2)
  println("sum ", 1 + 2)
  println("a", "b", "c")
}
```

```
OUTPUT
sum3
sum 3
abc
```

`print_color(color, value, ...)`

Prints a line wrapped in a bold bright ANSI colour escape sequence.

- **Aliases:** th: `พิมพ์สี` (further languages exist)
- **Parameters:** `color` - number 0..7 (clamped), mapped to bright foreground colours: 0 black, 1 red, 2 green, 3 yellow, 4 blue, 5 magenta, 6 cyan, 7 white. The remaining arguments are joined without separator.
- **Returns:** `()`.
- **Note:** Writes the raw escape codes `ESC[1;9Nm ... ESC[0m`; redirect the output to a file and they appear as literal control characters. On Windows the console is switched to virtual-terminal mode automatically.

```
GUI
bind start = do {
  print_color(2, "ok: ", "all tests passed")
  print_color(1, "error!")
}
```

`print_file(text)`

Writes text to standard output **without** a trailing newline.

- **Aliases:** th: `พิมพ์ไฟล์` (further languages exist)

- **Parameters:** `text` - converted to text; only the first argument is used.
- **Returns:** `()`.
- **Note:** Despite its name it does not touch the filesystem. It is the only way to print without a newline.

```
bind start = do {
  print_file("no newline")
  print_file(" - still same line")
  print("")
}
```

```
OUTPUT
no newline - still same line
```

`format(template, value, ...)`

Builds a string by substituting each `{}` in `template` with the next argument.

- **Aliases:** zh: `格式` ; ja: `フォーマット` ; ko: `서식` ; th: `รูปแบบ` (further languages exist)
- **Parameters:** `template` - text containing `{}` placeholders. `{:.N}` formats the next argument (when it is a number) with N decimal places. Remaining arguments fill the placeholders in order.
- **Returns:** Text.
- **Note:** A `{}` with no matching argument is dropped (becomes empty). Extra arguments are ignored.
- **Note:** Only `{}` and `{:.N}` are supported; `{:5}`, `{:>8}`, `{0}` and `{name}` are not. A literal brace cannot be escaped.
- **Note:** If the first argument is not text, it is simply converted to text and returned.

```
bind start = do {
  println(format("x={} y={}", 1, "two"))
  println(format("pi is about {:.3}", 3.14159265))
  println(format("{} and {}", "only one"))
  println(format("[{}]", [1, 2]))
}
```

```
OUTPUT
x=1 y=two
pi is about 3.142
only one and
[[1, 2]]
```

`ok(value)`

Wraps a value as a success result.

- **Aliases:** zh: `好`; ja: `良し`; ko: `좋아`; th: `โอเค` (further languages exist)
- **Parameters:** `value` - any value (default `()`).
- **Returns:** A result that prints as `Ok(value)`.
- **Note:** Use `match r { ok(v) => ..., bad(e) => ... }` to unpack. Calling any method on a result returns the inner value (`ok(5).anything` is `5`), which is rarely what you want; prefer `match`.

```
fn half(n) {
  if n % 2 == 0 { ok(n / 2) } else { bad("odd number") }
}
bind start = do {
  println(half(10))
  println(match half(10) { ok(v) => v, bad(e) => e })
}
```

```
OUTPUT
Ok(5)
5
```

`bad(reason)`

Wraps a value as a failure result.

- **Aliases:** English: `err`; zh: `坏`; ja: `悪い`; ko: `나쁨`; th: `ผิด` (further languages exist)
- **Parameters:** `reason` - any value, usually a message string.
- **Returns:** A result that prints as `Err(reason)`.

- **Note:** The English builtin also answers to `err`. Note the printed form is `Err(...)`, while the pattern and constructor are spelled `bad`.

```
fn half(n) {
  if n % 2 == 0 { ok(n / 2) } else { bad("odd number") }
}
bind start = do {
  println(half(7))
  println(match half(7) { ok(v) => "value", bad(e) => e })
}
```

```
OUTPUT
Err(odd number)
odd number
```

`try(function, arg, ...)`

Calls a closure and turns a runtime error inside it into a `bad` result instead of stopping the program.

- **Aliases:** English: `try_call` (further languages exist)
- **Parameters:** `function` - a closure or function value; the remaining arguments are passed to it.
- **Returns:** `ok(result)` on success, `bad(message)` if the call raised a runtime error, or `bad("try: first argument must be a function")` if the first argument is not callable.
- **Note:** Closures with no parameters must be written `| | expr` (with a space): the lexer reads `||` as the boolean "or" operator.
- **Note:** This example runs correctly only on the interpreter. The default JIT backend lowers closures passed as call arguments to `()`, so `try` reports that its argument is not a function.
- **Note:** A panic such as stack overflow is not caught; only ordinary runtime errors are.

INTERPRETER ONLY

```
bind start = do {
  println(try(| | 10 / 2))
  println(try(| | read_file("no/such/file.txt")))
  println(try(| a, b | a + b, 3, 4))
}
```

OUTPUT

```
Ok(5)
Err(read_file 'no/such/file.txt': The system cannot find the path specified. (os error 3))
Ok(7)
```

Run this example with `ling run --interp`; the default JIT backend prints:

OUTPUT

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

Numbers and math

All numbers are 64-bit floating point. Whole values print without a decimal point. Angles are in radians. Missing numeric arguments default to `0` (or the neutral value noted below) rather than raising an error.

Name	Summary
`sqrt(x)`	Square root
`pow(base, exponent)`	Raises `base` to `exponent` (floating-point power)
`abs(x)`	Absolute value
`floor(x)`	Rounds down toward negative infinity
`ceil(x)`	Rounds up toward positive infinity
`round(x)`	Rounds to the nearest whole number; halves round away from zero
`trunc(x)`	Discards the fractional part (rounds toward zero)
`fract(x)`	Fractional part of a number (`x - trunc(x)`)

Name	Summary
`min(a, b)`	Smaller of two numbers
`max(a, b)`	Larger of two numbers
`clamp(x, low, high)`	Limits `x` to the range `low..high`
`sign(x)`	Sign of a number: `1` for positive numbers **and zero**, `-1` for negative
`sin(x)`	Sine of an angle in radians
`cos(x)`	Cosine of an angle in radians
`tan(x)`	Tangent of an angle in radians
`asin(x)`	Inverse sine
`acos(x)`	Inverse cosine
`atan(x)`	Inverse tangent
`atan2(y, x)`	Angle of the point `(x, y)` from the positive x axis
`tanh(x)`	Hyperbolic tangent
`exp(x)`	e raised to the power `x`
`ln(x)`	Natural logarithm
`log2(x)`	Base-2 logarithm
`log10(x)`	Base-10 logarithm
`cbrt(x)`	Cube root
`hypot(x, y)`	Length of the hypotenuse, `sqrt(x*x + y*y)`
`pi()`	The constant pi (3.14159...)
`tau()`	The constant tau, equal to 2 * pi
`lerp(a, b, t)`	Linear interpolation: `a + (b - a) * t`
`smoothstep(low, high, x)`	Smooth Hermite interpolation: 0 below `low`, 1 above `high`, an S-curve between
`vnoise(x, y, seed)`	Smooth 2D value noise
`fbm(x, y, octaves, seed)`	Fractal (multi-octave) value noise
`perlin(x, y, z)`	Classic 3D Perlin gradient noise

`sqrt(x)`

Square root.

- **Aliases:** zh: `根`; ja: `平方根`; ko: `제곱근`; th: `รากที่สอง` (further languages exist)

- **Parameters:** `x` - number.
- **Returns:** Number (`NaN` for negative `x`).

```
bind start = do {
  println(sqrt(16))
  println(sqrt(2))
}
```

```
OUTPUT
4
1.4142135623730951
```

`pow(base, exponent)`

Raises `base` to `exponent` (floating-point power).

- **Aliases:** zh: `幂` ; ja: `べき乗` ; ko: `거듭제곱` ; th: `ยกกำลัง` (further languages exist)
- **Parameters:** `base` , `exponent` - numbers (exponent defaults to 1).
- **Returns:** Number.

```
bind start = do {
  println(pow(2, 10))
  println(pow(9, 0.5))
}
```

```
OUTPUT
1024
3
```

`abs(x)`

Absolute value.

- **Aliases:** zh: `绝对值` , `绝对` ; ja: `絶対値` ; ko: `절댓값` , `절대값` ; th: `ค่าสัมบูรณ์` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```ling
bind start = do {
  println(abs(0 - 7))
  println(abs(2.5))
}
```

```
OUTPUT
7
2.5
```

`floor(x)`

Rounds down toward negative infinity.

- **Aliases:** zh: `向下取整`, `下整`, `床関数`; ko: `내림`; th: `ปัดลง` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```ling
bind start = do {
  println(floor(2.7))
  println(floor(0 - 2.1))
}
```

```
OUTPUT
2
-3
```

`ceil(x)`

Rounds up toward positive infinity.

- **Aliases:** zh: `向上取整`, `上整`, `天井関数`; ko: `올림`; th: `ปัดขึ้น` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```
bind start = do {
  println(ceil(2.1))
  println(ceil(0 - 2.9))
}
```

```
OUTPUT
3
-2
```

`round(x)`

Rounds to the nearest whole number; halves round away from zero.

- **Aliases:** zh: `四舍五入`, `四舍`; ja: `四捨五入`; ko: `반올림`; th: `ปัดเศษ` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```
bind start = do {
  println(round(2.5))
  println(round(2.4))
  println(round(0 - 2.5))
}
```

```
OUTPUT
3
2
-3
```

`trunc(x)`

Discards the fractional part (rounds toward zero).

- **Aliases:** English: `int`; zh: `取整`, `整数`, `截整`; ja: `整数化`, `切り捨て`; ko: `정수화`, `정수`, `버림`; th: `ตัดทศนิยม` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

- **Note:** `int` is an English alias, but it truncates and returns a number; it does not convert text. Use `to_number` for that.

```
bind start = do {
  println(trunc(2.9))
  println(trunc(0 - 2.9))
  println(int(7.99))
}
```

```
OUTPUT
2
-2
7
```

`fract(x)`

Fractional part of a number (`x - trunc(x)`).

- **Aliases:** zh: `小数部分`, `小数部`; ko: `소수부` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```
bind start = do {
  println(fract(2.25))
  println(fract(0 - 2.25))
}
```

```
OUTPUT
0.25
-0.25
```

`min(a, b)`

Smaller of two numbers.

- **Aliases:** English: `minimum`; zh: `最小`; ko: `최솟값`; th: `ต่ำสุด` (further languages exist)
- **Parameters:** `a`, `b` - numbers.

- **Returns:** Number.
- **Note:** Takes exactly two arguments. For a list use `list_min`.

```
bind start = do {
  println(min(3, 1))
}
```

```
OUTPUT
1
```

`max(a, b)`

Larger of two numbers.

- **Aliases:** English: `maximum`; zh: `最大`; ko: `최댓값`; th: `สูงสุด` (further languages exist)
- **Parameters:** `a`, `b` - numbers.
- **Returns:** Number.
- **Note:** Takes exactly two arguments. For a list use `list_max`.

```
bind start = do {
  println(max(3, 1))
}
```

```
OUTPUT
3
```

`clamp(x, low, high)`

Limits `x` to the range `low..high`.

- **Aliases:** zh: `截取`; ja: `範囲制限`; ko: `범위제한`; th: `จำกัด` (further languages exist)
- **Parameters:** `x`, `low` (default 0), `high` (default 1).
- **Returns:** Number.
- **Note:** If `low > high` the bounds are swapped instead of raising an error.

```
bind start = do {
  println(clamp(15, 0, 10))
  println(clamp(0 - 3, 0, 10))
  println(clamp(5, 0, 10))
}
```

```
OUTPUT
10
0
5
```

`sign(x)`

Sign of a number: `1` for positive numbers **and zero**, `-1` for negative.

- **Aliases:** zh: `符号` ; ja: `符号関数` ; ko: `부호` ; th: `เครื่องหมาย` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.
- **Note:** Follows IEEE-754 `signum` : `sign(0)` is `1` , not `0` .

```
bind start = do {
  println(sign(0 - 4))
  println(sign(9))
  println(sign(0))
}
```

```
OUTPUT
-1
1
1
```

`sin(x)`

Sine of an angle in radians.

- **Aliases:** zh: `正弦` ; ja: `サイン` ; ko: `사인` ; th: `ไซน์` (further languages exist)

- **Parameters:** `x` - radians.
- **Returns:** Number.

```
bind start = do {
  println(sin(0))
  println(sin(pi() / 2))
}
```

```
OUTPUT
0
1
```

`cos(x)`

Cosine of an angle in radians.

- **Aliases:** zh: `余弦` ; ja: `コサイン` ; ko: `코사인` ; th: `โคไซน์` (further languages exist)
- **Parameters:** `x` - radians.
- **Returns:** Number.

```
bind start = do {
  println(cos(0))
  println(cos(pi()))
}
```

```
OUTPUT
1
-1
```

`tan(x)`

Tangent of an angle in radians.

- **Aliases:** zh: `正切` ; ja: `タンジェント` ; ko: `탄젠트` ; th: `แทนเจนต์` (further languages exist)
- **Parameters:** `x` - radians.
- **Returns:** Number.

```
bind start = do {
  println(tan(0))
}
```

```
OUTPUT
0
```

`asin(x)`

Inverse sine.

- **Aliases:** English: `arcsin` ; zh: `反正弦` ; ja: `アークサイン` ; ko: `아크사인` ; th: `อาร์กไซน์` (further languages exist)
- **Parameters:** `x` - number in -1..1.
- **Returns:** Radians.

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

```
OUTPUT
1.5707963267948966
```

`acos(x)`

Inverse cosine.

- **Aliases:** English: `arccos` ; zh: `反余弦` ; ja: `アークコサイン` ; ko: `아크코사인` ; th: `อาร์กโคไซน์` (further languages exist)
- **Parameters:** `x` - number in -1..1.
- **Returns:** Radians.

```
bind start = do {
  println(acos(1))
  println(acos(0))
}
```

```
OUTPUT
0
1.57079632679948966
```

`atan(x)`

Inverse tangent.

- **Aliases:** English: `arctan` ; zh: `反正切` ; ja: `アークタンジェント` ; ko: `아크탄젠트` ; th: `อาร์กแทนเจนต์` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Radians.

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

```
OUTPUT
0.7853981633974483
```

`atan2(y, x)`

Angle of the point `(x, y)` from the positive x axis. Note the argument order: **y first**.

- **Aliases:** English: `arctan2` ; zh: `反正切2` ; ja: `アークタンジェント2` ; ko: `아크탄젠트2` (further languages exist)
- **Parameters:** `y` , `x` - numbers (x defaults to 1).
- **Returns:** Radians in -pi..pi.

```
bind start = do {
  println(atan2(1, 1))
  println(atan2(1, 0 - 1))
}
```

```
OUTPUT
0.7853981633974483
2.356194490192345
```

`exp(x)`

e raised to the power `x`.

- **Aliases:** zh: `指数`, `指数関数`; ko: `지수` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```
bind start = do {
  println(exp(0))
  println(exp(1))
}
```

```
OUTPUT
1
2.718281828459045
```

`ln(x)`

Natural logarithm.

- **Aliases:** English: `log`; zh: `对数`; ja: `対数`; ko: `로그`; th: `ลอการิทึม` (further languages exist)
- **Parameters:** `x` - number (default 1).
- **Returns:** Number.
- **Note:** `log` is an alias of `ln` (natural log), **not** base 10. Use `log10` and `log2` for those.

```
bind start = do {
  println(ln(1))
  println(log(exp(2)))
}
```

```
OUTPUT
0
2
```

`log2(x)`

Base-2 logarithm.

- **Aliases:** zh: `对数2`, `对数2`; ko: `로그2` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```
bind start = do {
  println(log2(8))
}
```

```
OUTPUT
3
```

`log10(x)`

Base-10 logarithm.

- **Aliases:** zh: `对数10`, `对数10`; ko: `로그10` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```
bind start = do {
  println(log10(1000))
}
```

```
OUTPUT
3
```

`cbrt(x)`

Cube root.

- **Aliases:** zh: `立方根` ; ko: `세제곱근` ; th: `รากที่สาม` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number.

```
bind start = do {
  println(cbrt(27))
  println(cbrt(0 - 8))
}
```

```
OUTPUT
3
-2
```

`hypot(x, y)`

Length of the hypotenuse, `sqrt(x*x + y*y)` .

- **Aliases:** zh: `斜边` , `斜辺` ; ko: `빗변` (further languages exist)
- **Parameters:** `x` , `y` - numbers.
- **Returns:** Number.

```
bind start = do {
  println(hypot(3, 4))
}
```

```
OUTPUT
5
```

`pi()`

The constant pi (3.14159…).

- **Aliases:** zh: `圆周率` ; ja: `円周率` ; ko: `파이` ; th: `พาย` (further languages exist)

- **Parameters:** none.
- **Returns:** Number.
- **Note:** Always call it with parentheses. As a bare identifier `pi` evaluates to `()` under the default JIT backend (the interpreter does resolve the bare name).

```
bind start = do {
  println(pi())
  println(2 * pi() == tau())
}
```

```
OUTPUT
3.141592653589793
true
```

`tau()`

The constant tau, equal to 2 * pi.

- **Aliases:** zh: `双周率` ; ja: `タウ` ; ko: `타우` ; th: `ทาว` (further languages exist)
- **Parameters:** none.
- **Returns:** Number.

```
bind start = do {
  println(tau())
}
```

```
OUTPUT
6.283185307179586
```

`lerp(a, b, t)`

Linear interpolation: `a + (b - a) * t`.

- **Aliases:** zh: `线性插值` ; ja: `線形補間` ; ko: `선형보간` ; th: `ค่าระหว่าง` (further languages exist)
- **Parameters:** `a` (default 0), `b` (default 1), `t` (default 0).
- **Returns:** Number.
- **Note:** `t` is not clamped.

```
bind start = do {
  println(lerp(0, 10, 0.25))
  println(lerp(10, 20, 1.5))
}
```

```
OUTPUT
2.5
25
```

`len(value)`

Number of characters in a string, or elements in a list.

- **Aliases:** English: `str_len`; zh: `长度`; ja: `長さ`; ko: `길이`; th: `ความยาว` (further languages exist)
- **Parameters:** `value` - text or list. Anything else gives `0`.
- **Returns:** Number.
- **Note:** The English builtin is `str_len`; `len` is its alias. The method form `"abc".len()` counts **bytes** (`"é".len()` is 2) while `len("é")` counts characters (1).

```
bind start = do {
  println(len("hello"))
  println(len("héllo"))
  println(len([10, 20, 30]))
  println(len(42))
}
```

```
OUTPUT
5
5
3
0
```

`to_str(value)`

Converts any value to its text form.

- **Aliases:** English: `str`, `num_str`; th: `แปลงสตริง` (further languages exist)

- **Parameters:** `value` - any value.
- **Returns:** Text.

```
bind start = do {
  println(to_str(12) + "!")
  println(str(3.5))
  println(to_str([1, 2]))
  println(len(to_str(1000)))
}
```

```
OUTPUT
12!
3.5
[1, 2]
4
```

`to_number(text)`

Parses text as a decimal number.

- **Aliases:** zh: `转数字` (further languages exist)
- **Parameters:** `text` - leading and trailing whitespace is ignored.
- **Returns:** Number; `0` **if the text is not a number** (no error).
- **Note:** Cannot distinguish `"0"` from invalid input. Hex, exponents and underscores are not recognised; see `hex_to_number`.

```
bind start = do {
  println(to_number("42") + 1)
  println(to_number("3.5"))
  println(to_number("abc"))
  println(to_number("  7  "))
}
```

```
OUTPUT
43
3.5
0
7
```

`hex_to_number(text)`

Parses a hexadecimal string (no `0x` prefix) as a whole number.

- **Aliases:** zh: `十六进制转数字` (further languages exist)
- **Parameters:** `text` - hex digits.
- **Returns:** Number; `0` on failure.

```
bind start = do {
  println(hex_to_number("ff"))
  println(hex_to_number("1F40"))
  println(hex_to_number("zz"))
}
```

```
OUTPUT
255
8000
0
```

`split(text, separator)`

Splits text at every occurrence of a separator.

- **Aliases:** English: `str_split`; th: `แยก` (further languages exist)
- **Parameters:** `text`; `separator` (default newline; an empty separator also means newline).
- **Returns:** List of strings.
- **Note:** Splitting never trims; consecutive separators produce empty strings.

```
bind start = do {
  println(split("a,b,c", ","))
  println(len(split("a,,b", ",")))
  println(split("line1\nline2"))
}
```

```
OUTPUT
[a, b, c]
3
[line1, line2]
```

`trim(text)`

Removes whitespace from both ends.

- **Aliases:** English: `str_trim`; th: `ตัดช่องว่าง` (further languages exist)
- **Parameters:** `text`.
- **Returns:** Text.

```
bind start = do {
  println("[" + trim("  hi  ") + "]")
}
```

```
OUTPUT
[hi]
```

`starts_with(text, prefix)`

True if `text` begins with `prefix`.

- **Aliases:** English: `str_starts_with`; th: `เริ่มด้วย` (further languages exist)
- **Parameters:** `text`, `prefix`.
- **Returns:** Boolean.

```
bind start = do {
  println(starts_with("hello", "he"))
  println(starts_with("hello", "lo"))
}
```

```
OUTPUT
true
false
```

`ends_with(text, suffix)`

True if `text` ends with `suffix`.

- **Aliases:** English: `str_ends_with`; th: `ลงท้ายด้วย` (further languages exist)
- **Parameters:** `text`, `suffix`.
- **Returns:** Boolean.

```
bind start = do {
  println(ends_with("report.pdf", ".pdf"))
}
```

```
OUTPUT
true
```

`str_replace(text, from, to)`

Replaces **every** occurrence of `from` with `to`.

- **Aliases:** th: `แทนสตริง` (further languages exist)
- **Parameters:** `text`, `from`, `to`.
- **Returns:** Text.

```
bind start = do {
  println(str_replace("a-b-c", "-", "+"))
}
```

```
OUTPUT
a+b+c
```

`str_find(text, needle)`

Position of the first occurrence of `needle`.

- **Aliases:** th: `หาในสตริง` (further languages exist)
- **Parameters:** `text`, `needle`.
- **Returns:** Character index (0-based), or `-1` if absent.

```
bind start = do {
  println(str_find("hello", "l"))
  println(str_find("hello", "z"))
}
```

```
OUTPUT
2
-1
```

`substr(text, start, length)`

Extracts a slice by character position.

- **Aliases:** English: `str_slice`; th: `ส่วนสตริง` (further languages exist)
- **Parameters:** `text`; `start` (0-based); `length` (optional; default: to the end).
- **Returns:** Text. Out-of-range values are clamped, never an error.

```
bind start = do {
  println(substr("hello", 1, 3))
  println(substr("hello", 2))
  println(substr("hello", 10, 2) + "|")
}
```

```
OUTPUT
ell
llo
|
```

`str_repeat(text, count)`

Repeats a string.

- **Aliases:** th: `ทำซ้ำสตริง` (further languages exist)
- **Parameters:** `text`; `count` (default 1).
- **Returns:** Text.

```
bind start = do {
  println(str_repeat("ab", 3))
  println(str_repeat("=", 10))
}
```

```
OUTPUT
ababab
==========
```

`str_upper(text)`

Converts to upper case.

- **Aliases:** none (English spelling only)
- **Parameters:** `text`.
- **Returns:** Text.

```
bind start = do {
  println(str_upper("hello"))
}
```

```
OUTPUT
HELLO
```

`str_lower(text)`

Converts to lower case.

- **Aliases:** none (English spelling only)
- **Parameters:** `text`.
- **Returns:** Text.

```
bind start = do {
  println(str_lower("HeLLo"))
}
```

```
OUTPUT
hello
```

`str_strip_prefix(text, prefix)`

Removes a prefix if present, otherwise returns the text unchanged.

- **Aliases:** th: `ตัดคำนำหน้า` (further languages exist)
- **Parameters:** `text`, `prefix`.
- **Returns:** Text.

```
bind start = do {
  println(str_strip_prefix("Bearer abc", "Bearer "))
  println(str_strip_prefix("abc", "xyz"))
}
```

```
OUTPUT
abc
abc
```

`.len()`

Length of a string (in **bytes**) or list (elements).

- **Aliases:** zh: `长`, `长度`; ja: `長さ`; ko: `길이`; th: `ความยาว`

- **Parameters:** receiver: text or list.
- **Returns:** Number.
- **Note:** For text, use the builtin `len(text)` when you want characters.

```ling
bind start = do {
  println([1, 2, 3].len())
  println("abc".len())
  println("é".len())
  println(len("é"))
}
```

```
OUTPUT
3
3
2
1
```

`.push(value)`

Returns a copy of the list with `value` appended.

- **Aliases:** zh: `推`, `添加`; ja: `追加`; ko: `추가`; th: `เพิ่ม`
- **Parameters:** receiver: list; `value`.
- **Returns:** New list.
- **Note:** The original list is unchanged under the interpreter, so write `bind xs = xs.push(4)`. Under the default JIT backend the original may be updated in place as well; do not rely on either.

```ling
bind start = do {
  bind xs = [1, 2]
  bind xs = xs.push(3)
  println(xs)
}
```

```
OUTPUT
[1, 2, 3]
```

`.contains(text)`

True if a string contains a substring.

- **Aliases:** zh: `包含`
- **Parameters:** receiver: text; `text`.
- **Returns:** Boolean (false for non-string argument).

```
bind start = do {
  println("hello".contains("ell"))
  println("hello".contains("z"))
}
```

```
OUTPUT
true
false
```

`.is_empty()`

True if a string has no characters.

- **Aliases:** zh: `是空`
- **Parameters:** receiver: text.
- **Returns:** Boolean.

```
bind start = do {
  println("".is_empty())
  println("x".is_empty())
}
```

```
OUTPUT
true
false
```

`.to_string()`

Returns the string itself.

- **Aliases:** zh: `转文`
- **Parameters:** receiver: text.
- **Returns:** Text.

```
bind start = do {
  println("abc".to_string())
}
```

```
OUTPUT
abc
```

`.push_str(text)`

Returns the string with `text` appended (a copy, like `+`).

- **Aliases:** zh: `推_文`
- **Parameters:** receiver: text; `text`.
- **Returns:** Text.

```
bind start = do {
  println("abc".push_str("def"))
}
```

```
OUTPUT
abcdef
```

`.tag`

Name of the variant of a `choose` (enum) value, as text.

- **Aliases:** zh: `标签` ; ja: `タグ` ; ko: `태그` ; th: `ป้าย`
- **Parameters:** receiver: enum value (written without parentheses).
- **Returns:** Text.

```
choose Shape { Circle(r), Dot }
bind start = do {
  println(Circle(2).tag)
  println(Dot().tag)
}
```

```
OUTPUT
Circle
Dot
```

`.name`

Reads a field of a `form` (struct) value.

- **Aliases:** none (the field name is whatever you declared)
- **Parameters:** receiver: struct value; the field name follows the dot.
- **Returns:** The field value. Error `no field ...` if it does not exist.

```
form Point { x, y }
bind start = do {
  bind p = Point(3, 4)
  println(p.x)
  println(p.y)
}
```

```
OUTPUT
3
4
```

Lists

Lists are ordered, mixed-type sequences written `[1, "two", 3.0]`. They behave as values: functions return a new list instead of mutating. Because rebinding is the only form of assignment, the idiom is `bind xs = list_push(xs, item)`. Indexing is `xs[i]` (zero-based); reading past the end is a runtime error under the interpreter.

Name	Summary
`list_new()`	Creates an empty list
`list_push(list, value)`	Returns the list with `value` appended
`list_get(list, index)`	Reads an element by zero-based index
`list_set(list, index, value)`	Returns the list with the element at `index` replaced
`list_max(list, default)`	Largest number in a list
`list_min(list, default)`	Smallest number in a list
`list_join(list, separator)`	Joins the elements into one string
`list_map(list, function)`	Applies a function to every element
`list_filter(list, predicate)`	Keeps the elements for which the predicate returns `true`
`list_find(list, predicate)`	First element for which the predicate returns `true`

`list_new()`

Creates an empty list.

- **Aliases:** English: `Vec::with_capacity(n)`, `Vec::from(list)`; zh: `新建列表`; ja: `新規リスト`; ko: `새목록`; th: `รายการใหม่` (further languages exist)
- **Parameters:** none.
- **Returns:** Empty list.

```
bind start = do {
  bind xs = list_new()
  println(xs)
  println(len(xs))
}
```

```
OUTPUT
[]
0
```

`list_push(list, value)`

Returns the list with `value` appended.

- **Aliases:** zh: `列表添加`; ja: `リスト追加`; ko: `목록추가`; th: `เพิ่มรายการ` (further languages exist)

- **Parameters:** `list`; `value`.
- **Returns:** The extended list.
- **Note:** Always rebind the result: `bind xs = list_push(xs, 5)`. The interpreter optimises exactly this pattern to run in place.
- **Note:** If the first argument is not a list, a new one-element list is returned.

```
bind start = do {
  bind xs = list_new()
  bind xs = list_push(xs, 10)
  bind xs = list_push(xs, "twenty")
  println(xs)
}
```

```
OUTPUT
[10, twenty]
```

`list_get(list, index)`

Reads an element by zero-based index.

- **Aliases:** zh: `取元素`, `要素取得`; ko: `요소가져오기`; th: `รับรายการ` (further languages exist)
- **Parameters:** `list`; `index` (default 0).
- **Returns:** The element; **an empty string `""` if the index is out of range** (no error).
- **Note:** `xs[i]` is the stricter form: it is a runtime error when out of range under the interpreter.

```
bind start = do {
  bind xs = [10, 20, 30]
  println(list_get(xs, 1))
  println("[" + list_get(xs, 9) + "]")
}
```

```
OUTPUT
20
[]
```

`list_set(list, index, value)`

Returns the list with the element at `index` replaced.

- **Aliases:** zh: `设元素`, `要素設定`; ko: `요소설정`; th: `ตั้งรายการ` (further languages exist)
- **Parameters:** `list`; `index`; `value`.
- **Returns:** Updated list. An out-of-range index leaves it unchanged.
- **Note:** Rebind the result: `bind xs = list_set(xs, 0, 99)`.

```
bind start = do {
  bind xs = [1, 2, 3]
  bind xs = list_set(xs, 0, 99)
  println(xs)
}
```

```
OUTPUT
[99, 2, 3]
```

`list_max(list, default)`

Largest number in a list.

- **Aliases:** zh: `列表最大值` (further languages exist)
- **Parameters:** `list` of numbers; `default` returned for an empty list (default 0).
- **Returns:** Number.

```
bind start = do {
  println(list_max([3, 9, 2], 0))
  println(list_max([], 0 - 1))
}
```

```
OUTPUT
9
-1
```

`list_min(list, default)`

Smallest number in a list.

- **Aliases:** zh: `列表最小值` (further languages exist)
- **Parameters:** `list` of numbers; `default` returned for an empty list (default 0).
- **Returns:** Number.

```
bind start = do {
  println(list_min([3, 9, 2], 0))
}
```

```
OUTPUT
2
```

`list_join(list, separator)`

Joins the elements into one string.

- **Aliases:** English: `join`; zh: `连接`, `連結`; ko: `연결`; th: `รวมรายการ` (further languages exist)
- **Parameters:** `list`; `separator` (default empty).
- **Returns:** Text.

```
bind start = do {
  println(list_join(["a", "b", "c"], "-"))
  println(join([1, 2, 3], ", "))
}
```

```
OUTPUT
a-b-c
1, 2, 3
```

`list_map(list, function)`

Applies a function to every element.

- **Aliases:** zh: `映射列表` (further languages exist)

- **Parameters:** `list`; `function` - closure taking one argument.
- **Returns:** New list of the results.
- **Note:** Closure arguments only work on the interpreter (`ling run --interp`); the default JIT backend passes `()` instead of the closure.

INTERPRETER ONLY

```
bind start = do {
  println(list_map([1, 2, 3], |x| x * x))
}
```

OUTPUT

```
[1, 4, 9]
```

Run this example with `ling run --interp`; the default JIT backend prints:

OUTPUT

```
()
```

`list_filter(list, predicate)`

Keeps the elements for which the predicate returns `true`.

- **Aliases:** zh: `过滤列表` (further languages exist)
- **Parameters:** `list`; `predicate` - closure returning a boolean.
- **Returns:** New list.
- **Note:** Only a literal `true` keeps an element; truthy numbers do not. Interpreter only (see `list_map`).

INTERPRETER ONLY

```
bind start = do {
  println(list_filter([1, 2, 3, 4, 5, 6], |x| x % 2 == 0))
}
```

OUTPUT

```
[2, 4, 6]
```

Run this example with `ling run --interp`; the default JIT backend prints:

```
OUTPUT
()
```

`list_find(list, predicate)`

First element for which the predicate returns `true`.

- **Aliases:** zh: `查找列表` (further languages exist)
- **Parameters:** `list`; `predicate` - closure.
- **Returns:** The element, or `()` if none matches.
- **Note:** Interpreter only (see `list_map`).

```
INTERPRETER ONLY
bind start = do {
  println(list_find([1, 5, 8, 12], |x| x > 6))
  println(list_find([1, 2], |x| x > 6))
}
```

```
OUTPUT
8
()
```

Run this example with `ling run --interp`; the default JIT backend prints:

```
OUTPUT
()
()
```

Files and paths

File builtins use forward slashes; backslashes in arguments are converted. Most failure modes return `false`, `0` or an empty value rather than raising an error; `read_file` is the exception and raises a runtime error for a missing file under the interpreter. Paths are relative to the current working directory.

Name	Summary
`read_file(path)`	Reads an entire text file
`write_file(path, text)`	Writes text to a file, creating parent folders and replacing existing content
`file_exists(path)`	True if the path exists (file or folder)
`is_file(path)`	True if the path is an existing regular file
`is_dir(path)`	True if the path is an existing directory
`list_dir(path)`	Lists the entries of a directory, sorted
`make_dir(path)`	Creates a directory and any missing parents
`remove_file(path)`	Deletes a file
`remove_dir_all(path)`	Recursively deletes a directory tree
`rename_file(from, to)`	Moves or renames a file (falls back to copy-and-delete across drives)
`copy_file(from, to)`	Copies a file (binary-safe), creating the destination folder
`file_size(path)`	Size of a file in bytes
`file_modified(path)`	Last-modified time as Unix seconds
`file_created(path)`	Creation time as Unix seconds (modification time on filesystems without one)
`path_name(path)`	The final component of a path (file name)
`path_ext(path)`	File extension, lower-cased, without the dot
`absolute_path(path)`	Resolves a path against the current working directory, using forward slashes
`file_write_bytes(path, bytes, append)`	Writes raw bytes (the text file builtins cannot)
`file_append_file(dest, source)`	Appends the contents of one file to another
`file_magic(path)`	Classifies a file by its first bytes

`read_file(path)`

Reads an entire text file.

- **Aliases:** th: `อ่านไฟล์` (further languages exist)
- **Parameters:** `path` - file path.
- **Returns:** Text (UTF-8). A missing or unreadable file is a runtime error (`read_file 'path': ...`).

- **Note:** Under the default JIT backend a failed read yields `()` instead of an error, so test with `file_exists` first.

```
bind start = do {
  write_file("note.txt", "line one\nline two")
  println(read_file("note.txt"))
  println(len(split(read_file("note.txt"), "\n")))
  remove_file("note.txt")
}
```

```
OUTPUT
line one
line two
2
```

`write_file(path, text)`

Writes text to a file, creating parent folders and replacing existing content.

- **Aliases:** th: `เขียนไฟล์` (further languages exist)
- **Parameters:** `path`; `text`.
- **Returns:** `()`. Failure to write is a runtime error.

```
bind start = do {
  write_file("out/data.txt", "hello")
  println(file_exists("out/data.txt"))
  remove_dir_all("out")
}
```

```
OUTPUT
true
```

`file_exists(path)`

True if the path exists (file or folder).

- **Aliases:** zh: `文件存在` (further languages exist)

- **Parameters:** `path`.
- **Returns:** Boolean.

```
bind start = do {
  println(file_exists("definitely_missing.txt"))
}
```

```
OUTPUT
false
```

`is_file(path)`

True if the path is an existing regular file.

- **Aliases:** th: `เป็นไฟล์` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean.

```
bind start = do {
  write_file("t.txt", "x")
  println(is_file("t.txt"))
  println(is_file("."))
  remove_file("t.txt")
}
```

```
OUTPUT
true
false
```

`is_dir(path)`

True if the path is an existing directory.

- **Aliases:** th: `เป็นไดเรกทอรี` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean.

```
bind start = do {
  println(is_dir("."))
  println(is_dir("missing_dir"))
}
```

```
OUTPUT
true
false
```

`list_dir(path)`

Lists the entries of a directory, sorted.

- **Aliases:** th: `รายการไดเรกทอรี` (further languages exist)
- **Parameters:** `path` (default `.`).
- **Returns:** List of path strings (each prefixed with the given path). Empty list if unreadable.

```
bind start = do {
  make_dir("demo")
  write_file("demo/b.txt", "")
  write_file("demo/a.txt", "")
  println(list_dir("demo"))
  remove_dir_all("demo")
}
```

```
OUTPUT
[demo/a.txt, demo/b.txt]
```

`make_dir(path)`

Creates a directory and any missing parents.

- **Aliases:** th: `สร้างไดเรกทอรี` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean success.

```
bind start = do {
  println(make_dir("a/b/c"))
  println(is_dir("a/b/c"))
  remove_dir_all("a")
}
```

```
OUTPUT
true
true
```

`remove_file(path)`

Deletes a file.

- **Aliases:** zh: `删除文件`; ja: `ファイル削除`; ko: `파일삭제`; th: `ลบไฟล์` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean; false if it could not be deleted (for example it does not exist).

```
bind start = do {
  write_file("tmp.txt", "x")
  println(remove_file("tmp.txt"))
  println(remove_file("tmp.txt"))
}
```

```
OUTPUT
true
false
```

`remove_dir_all(path)`

Recursively deletes a directory tree.

- **Aliases:** th: `ลบไดเรกทอรีทั้งหมด` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean; a path that does not exist counts as success.
- **Note:** Irreversible. Double-check the argument.

```
bind start = do {
  make_dir("scratch/inner")
  println(remove_dir_all("scratch"))
  println(is_dir("scratch"))
}
```

```
OUTPUT
true
false
```

`rename_file(from, to)`

Moves or renames a file (falls back to copy-and-delete across drives).

- **Aliases:** zh: `重命名文件`; ja: `ファイル改名`; ko: `파일이름변경`; th: `เปลี่ยนชื่อไฟล์` (further languages exist)
- **Parameters:** `from`, `to`.
- **Returns:** Boolean success.

```
bind start = do {
  write_file("old.txt", "data")
  println(rename_file("old.txt", "new.txt"))
  println(file_exists("old.txt"))
  println(read_file("new.txt"))
  remove_file("new.txt")
}
```

```
OUTPUT
true
false
data
```

`copy_file(from, to)`

Copies a file (binary-safe), creating the destination folder.

- **Aliases:** th: `คัดลอกไฟล์` (further languages exist)

- **Parameters:** `from`, `to`.
- **Returns:** Boolean success.

```
bind start = do {
  write_file("src.txt", "abc")
  println(copy_file("src.txt", "dst/copy.txt"))
  println(read_file("dst/copy.txt"))
  remove_file("src.txt")
  remove_dir_all("dst")
}
```

```
OUTPUT
true
abc
```

`file_size(path)`

Size of a file in bytes.

- **Aliases:** th: `ขนาดไฟล์` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Number; `0` if the file does not exist.

```
bind start = do {
  write_file("s.txt", "12345")
  println(file_size("s.txt"))
  remove_file("s.txt")
}
```

```
OUTPUT
5
```

`path_name(path)`

The final component of a path (file name).

- **Aliases:** th: `ชื่อไฟล์` (further languages exist)

- **Parameters:** `path`.
- **Returns:** Text; empty if none.

```
bind start = do {
  println(path_name("assets/img/logo.png"))
}
```

```
OUTPUT
logo.png
```

`path_ext(path)`

File extension, lower-cased, without the dot.

- **Aliases:** th: `นามสกุลไฟล์` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Text; empty if none.

```
bind start = do {
  println(path_ext("assets/img/Logo.PNG"))
  println("[" + path_ext("Makefile") + "]")
}
```

```
OUTPUT
png
[]
```

`absolute_path(path)`

Resolves a path against the current working directory, using forward slashes.

- **Aliases:** th: `พาธสัมบูรณ์` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Text. An already absolute path is returned unchanged (slashes normalised).
- **Note:** Does not check that the path exists and does not resolve `..`.

```
bind start = do {
  println(starts_with(absolute_path("x.txt"), "x.txt"))
  println(ends_with(absolute_path("x.txt"), "/x.txt"))
}
```

```
OUTPUT
false
true
```

`get_args()`

Returns the whole process command line.

- **Aliases:** th: `รับอาร์กิวเมนต์` (further languages exist)
- **Parameters:** none.
- **Returns:** List of strings. For `ling run prog.ling a b` the list starts with the `ling` executable path, then `run`, then the script path, then your arguments.
- **Note:** Because the first elements belong to the `ling` launcher, index from the end or use `cli_arg` for named options.

```
bind start = do {
  bind a = get_args()
  println(len(a) >= 3)
  println(a[1])
}
```

```
OUTPUT
true
run
```

`cli_arg(name, default)`

Reads the value of a `--name value` command-line option.

- **Aliases:** th: `อาร์กิวเมนต์` (further languages exist)

- **Parameters:** `name` - option name without dashes; `default` - returned when absent (default empty).
- **Returns:** Text.
- **Note:** Only the `--name value` form is understood (two separate arguments).

```
bind start = do {
  println(cli_arg("port", "8080"))
}
```

```
OUTPUT
8080
```

`env_get(name, default)`

Reads an environment variable.

- **Aliases:** th: `รับตัวแปรแวดล้อม` (further languages exist)
- **Parameters:** `name`; `default` returned if unset (default empty).
- **Returns:** Text.

```
bind start = do {
  println(env_get("LING_SURELY_UNSET_VAR", "fallback"))
}
```

```
OUTPUT
fallback
```

`sleep(milliseconds)`

Pauses the program.

- **Aliases:** English: `sleep_ms`; zh: `睡眠`, `流水:睡眠`; ja: `眠る`, `スリープ`; ko: `잠자기`, `잠`; th: `หยุด`, `นอน` (further languages exist)
- **Parameters:** `milliseconds` - number.
- **Returns:** `()`.

```
bind start = do {
  sleep(50)
  println("done")
}
```

```
OUTPUT
done
```

`rand()`

Pseudo-random number in the range 0 up to (not including) 1.

- **Aliases:** zh: `随机` ; ja: `乱数` ; ko: `난수` ; th: `สุ่ม` (further languages exist)
- **Parameters:** none (arguments are ignored).
- **Returns:** Number.
- **Note:** There is no seeding function and no integer form: scale and floor it, e.g. `floor(rand() * 6) + 1` for a die.
- **Note:** Not suitable for security; use `random_hex` for unpredictable bytes.

```
bind start = do {
  bind r = rand()
  println(r >= 0 && r < 1)
  bind die = floor(rand() * 6) + 1
  println(die >= 1 && die <= 6)
}
```

```
OUTPUT
true
true
```

`time_now()`

Seconds elapsed since the program started.

- **Aliases:** zh: `当前时间` ; ja: `経過時間` ; ko: `현재시간` ; th: `เวลาปัจจุบัน` (further languages exist)
- **Parameters:** none.

- **Returns:** Number (fractional seconds).

```
bind start = do {
  bind t0 = time_now()
  sleep(20)
  println(time_now() - t0 >= 0.02)
}
```

```
OUTPUT
true
```

`epoch_now()`

Wall-clock time as seconds since 1 January 1970 UTC.

- **Aliases:** English: `datetime`, `now_unix`; zh: `现在时刻`, `現在時刻`; ko: `현재시각`; th: `เวลาโลก` (further languages exist)
- **Parameters:** none.
- **Returns:** Number (fractional).
- **Note:** `now_unix` returns the same value. There are no date-formatting builtins.

```
bind start = do {
  println(epoch_now() > 1700000000)
  println(now_unix() > 1700000000)
}
```

```
OUTPUT
true
true
```

`random_hex(bytes)`

Cryptographically secure random bytes, as lower-case hex.

- **Aliases:** ko: `무작위16진수` (further languages exist)
- **Parameters:** `bytes` - number of bytes (default 16).
- **Returns:** Text of length 2 * bytes.

```
bind start = do {
  println(len(random_hex(16)))
  println(random_hex(4) == random_hex(4))
}
```

```
OUTPUT
32
false
```

Encoding, escaping and hashing

Name	Summary
`url_decode(text)`	Decodes percent-escapes and turns `+` into a space
`html_escape(text)`	Escapes `&` `<` `>` `"` `'` as HTML entities
`json_escape(text)`	Escapes a string for use inside a JSON string literal (quote, backslash, newline, tab, control characters)
`base64_encode(text)`	Standard base64 encoding of a string's UTF-8 bytes
`query_param(query, name, default)`	Reads one parameter from a URL query string, url-decoding the value
`cookie_get(header, name, default)`	Reads one cookie from a `Cookie` header string
`base64_decode_to_file(base64, path)`	Decodes base64 and writes the bytes to a file (a `data:...; base64,` prefix is tolerated)

`url_decode(text)`

Decodes percent-escapes and turns `+` into a space.

- **Aliases:** zh: `网址解码` (further languages exist)
- **Parameters:** `text`.
- **Returns:** Text.

```
bind start = do {
  println(url_decode("hello%20world+again"))
  println(url_decode("caf%C3%A9"))
}
```

```
OUTPUT
hello world again
café
```

`html_escape(text)`

Escapes `& < > " '` as HTML entities.

- **Aliases:** th: `กันเอชทีเอ็มแอล` (further languages exist)
- **Parameters:** `text`.
- **Returns:** Text.

```
bind start = do {
  println(html_escape("<b>\"Tom\" & Jerry</b>"))
}
```

```
OUTPUT
&lt;b&gt;&quot;Tom&quot; &amp; Jerry&lt;/b&gt;
```

`json_escape(text)`

Escapes a string for use inside a JSON string literal (quote, backslash, newline, tab, control characters).

- **Aliases:** th: `หนีเจสัน` (further languages exist)
- **Parameters:** `text`.
- **Returns:** Text (without surrounding quotes).
- **Note:** Ling has no JSON parser or generator; build JSON by concatenation and use this to make values safe.

```
bind start = do {
  println("{\"msg\": \"" + json_escape("say \"hi\"\nbye") + "\"}")
}
```

```
OUTPUT
{"msg": "say \"hi\"\nbye"}
```

`base64_encode(text)`

Standard base64 encoding of a string's UTF-8 bytes.

- **Aliases:** ko: `base64인코딩` (further languages exist)
- **Parameters:** `text` .
- **Returns:** Text.

```
bind start = do {
  println(base64_encode("hello"))
}
```

```
OUTPUT
aGVsbG8=
```

`query_param(query, name, default)`

Reads one parameter from a URL query string, url-decoding the value.

- **Aliases:** th: `พารามิเตอร์` (further languages exist)
- **Parameters:** `query` - text such as `a=1&b=x%20y` ; `name` ; `default` (empty if omitted).
- **Returns:** Text.

```
bind start = do {
  println(query_param("a=1&b=x%20y", "b", "none"))
  println(query_param("a=1", "zz", "none"))
}
```

```
OUTPUT
x y
none
```

`cookie_get(header, name, default)`

Reads one cookie from a `Cookie` header string.

- **Aliases:** th: `รับคุกกี้` (further languages exist)
- **Parameters:** `header` - text such as `sid=abc; x=1`; `name`; `default`.
- **Returns:** Text.

```
bind start = do {
  println(cookie_get("sid=abc; x=1", "x", ""))
}
```

```
OUTPUT
1
```

CHAPTER 19

Command Line Tools: `ling` and `lingfu`

Two programs ship with Ling:

- `ling` compiles, runs, visualises and packages individual programs.
- `lingfu` (灵符, "spirit token") is the project manager: it scaffolds projects, runs their tests, manages the `Ling.toml` manifest, publishes to a package registry and rewrites a project into another human language.

Both are installed to the same directory (`~/.cargo/bin` for a Cargo install). `lingfu` finds `ling` next to itself, then on `PATH`, and delegates to it for running and building. The transcripts below were produced with Ling 2030.1.53 (`ling`) and lingfu 2030.1.19.

Transcripts marked `console` are real terminal sessions; long absolute paths have been shortened.

`ling`

```
OUTPUT
ling run [--interp|--wasm|--O0|--O1|--O2|--O3] <file.ling>
ling <file.ling>                    (shorthand for: ling run <file.ling>)
ling build <file.ling|dir> [--out dir] [--platform list] [--icon file] [--pack] [--aot]
ling ast [path] [--technical|--artwork|--ling|--all] [--out dir]
ling visualize <file.ling>
ling convert <asset> [-o out.ling] [--no-compression]
```

Running `ling` with no arguments, or with anything it does not recognise, prints the usage text and exits with status 0. There is no `--version` or `--help` flag; the first line of the usage text carries the version (`ling 2030.1.53 – The Omniglot Systems Language`). Subcommands that other tools sometimes expect (`check`, `fmt`, `translate`, `test`) do **not** exist in `ling`; see the remarks under `lingfu check`.

A source file may use any of the extensions `.ling`, `.灵`, `.靈`, `.령` or `.ลิง`.

`ling run`

Runs a program: it parses the file, finds the entry point (`bind start = do { ... }`; see the syntax chapter) and executes it.

Flag	Effect
(none)	Cranelift JIT backend (the default). Falls back to the interpreter for the whole program if the JIT cannot compile it.
`--interp`	Tree-walking interpreter, the semantic reference. Gives the best diagnostics.
`--wasm` (alias `--web`)	Builds the program to WebAssembly, serves it on a local web server and opens the browser. Not verified here.
`--O0` `--O1` `--O2` `--O3`	JIT optimisation level (default `--O3`). Use a lower level if a very large program takes too long to compile.

The flags may appear before or after the file name; the first argument with a Ling extension is the program. Any further arguments stay on the command line and are visible to the program: `get_args()` returns the whole command line (including `ling`, `run` and the script path) and `cli_arg("port", "8080")` reads a `--port 99` option.

```
OUTPUT
$ ling run args.ling --port 99 extra
99
6
```

Behaviour worth knowing:

- Programs that call `http_serve` or `lingtp_serve` are automatically run on the interpreter, because the JIT does not yet support closure arguments.
- A file whose entry point is missing is treated as a library and fails with `error[runtime]: no entry point`, exit status 1.
- For a non-English source file, `ling` writes `[detected language: Chinese (...)]` (or Thai, Japanese, ...) to **standard error** before running.
- A missing file gives `[ling] error: file does not exist: <path>` and status 1; no file at all prints `Usage: ling run [--wasm|--interp|--O0|--O1|--O2|--O3] <file.ling>` and status 1.
- Parse errors are reported as `error[parse]: ...` with status 1. Runtime errors are `error[runtime]: ...` followed by a `traceback`.

```
OUTPUT
$ cat hello.ling
fn sq(n) { n * n }
bind start = do {
  println(sq(7))
}
$ ling run hello.ling
49
$ ling hello.ling
49
$ ling run --interp --O0 hello.ling
49
$ ling run nofile.ling
[ling] error: file does not exist: nofile.ling
```

`ling build`

Compiles a program or a project into a distributable.

```
OUTPUT
ling build <file.ling | project-dir> [--out dir] [--platform list] [--icon file] [--
pack] [--aot]
```

Option	Meaning
`<target>`	A single `.ling` file, or a directory. A directory is read through its manifest (`灵符.toml` , `ลิงฟู.toml` , then `Ling.toml`); a bare directory falls back to `main.ling` , `start.ling` , `启.灵` , `เริ่ม.ลิง` , a `src` / `灵源` / `ต้นกำเนิด` subfolder, or any `.ling` file. Defaults to `.` .
`--out <dir>`	Output folder. Default `dist` ; each platform is written to a sub-folder such as `dist/web` or `dist/windows` .
`--platform <list>`	Comma-separated targets: `web` , `win` (`windows`), `lin` (`linux`), `mac` (`macos` , `darwin`), `macarm` (`mac-arm` , `apple-silicon`), `pios` (`pi` , `raspios` , `linux-arm64` , `arm64`), `rpi` , `kernel` (`bare`), `lingos` (`app` , `apps`), `lingos-rpi` . `all` means `win,lin,mac,web` . Anything else aborts with `unknown platform` .
`--icon <file>`	Application icon (`.svg` , `.png` , `.jpg` , `.bmp` , `.ico`), overriding the manifest `icon` . Packed into a multi-resolution `.ico` for Windows executables; a missing file warns and falls back to the Ling logo.
`--pack`	Embed the `[includes]` resources (and the entry program) inside the executable so it is one self-contained file. They are unpacked to a per-application temporary folder at start-up.

Option	Meaning
`--aot`	Compile the program ahead of time to native machine code instead of embedding the interpreter.

If `--platform` is omitted the build uses the manifest's `targets` list, or, when that is empty, the host platform plus `web`.

Native targets are produced by generating and compiling a small Rust project under `.ling-build/` (or `灵碑/` for lingfu manifests), so a **Rust toolchain must be installed** and a first build can take minutes. The `web` target is built through the companion `lingc` binary. In the environment used for this reference a native build was started but aborted by a local linker failure, so the end-to-end result of `ling build` was **not verified**; its argument handling and manifest rules are documented from the source.

Manifest keys read by `ling build`

`ling build` reads its manifest with a simple **line-based** reader, not a full TOML parser. Consequences: every setting must be on one line, written `key = value` with a space before the `=`; section headings other than `[includes]` / `[包含]` and `[app.<name>]` are ignored for key lookup (the keys are found wherever they appear); and `targets`, `includes` and `ling-features` arrays must be written on a single line (except for the `[includes]` section form, one pattern per line).

Key (`ling.toml`)	`灵.toml`	Meaning
`name`	`名`	Project name (sanitised for use as an executable name).
`version`	`版`	Version string. Default `0.1.0`.
`entry`	*(auto-detected)*	Entry program, relative to the manifest. Default `main.ling`; the build stops with `entry '...' not found` if it does not exist.
`kind`	`型`	`bin` (default), `web`, `game`, `ui`, `ai`, `crypto`, `lib`, `polyglot`, `kernel`; also the localised words 网灵, 游灵, 显灵, 智灵, 密灵, 共修, 万言, 内核.
`icon`	`图标` (or `icon`)	Icon file relative to the project.
`targets`	`targets`	Default platform list, e.g. `targets = ["win", "lin", "web"]`.
`pack`	-	`pack = true` is the same as `--pack`.
`includes`	`包含`	Resource globs: `includes = ["music/*.wav", "font/*.otf"]`, or an `[includes]` / `[包含]` section with one quoted pattern per line. `*` matches within a path segment, `**` across segments, `?` one character; a bare folder name includes everything under it; a leading `/` is optional.

Key (`Ling.toml`)	`[project]` `#.toml`	Meaning
`ling-features`	-	Cargo features of the `ling` crate to enable in the built executable, e.g. `["web"]`. Required for programs that use feature-gated builtins such as the SQLite functions (`db_open`, `db_exec`, `db_query` need the `web` feature; without it the program fails with `unknown function 'db_open'`).
`portable`	-	Folder name: desktop builds run `<folder>/<entry>` on the interpreter from beside the executable, keeping multi-folder layouts that `use` sub-files intact.
`graphics`, `embed_root`, `kernel_app`, `bundle_app`, `entry_os`, `includes_os`, `[app.<name>]`	-	Options for the `kernel`, `rpi`, `lingos` targets (framebuffer request, embedding roots, built-in applications). Used by the LingOS images.

Included resources are copied next to the executable in the same relative layout, or embedded with `--pack`. After a `--pack` build the usual file builtins (`read_file`, `font_load`, ...) work unchanged because the working directory is switched to the unpacked folder.

```toml
# Ling.toml
[project]
name = "my-game"
version = "1.0.0"
entry = "main.ling"
icon = "images/logo.svg"
targets = ["win", "lin", "web"]
pack = true

[includes]
"/music/*.wav"
"/font/*.otf"
"/levels/**/*.json"
```

`ling ast`

Draws the abstract syntax tree of a file or a whole project as SVG. A directory is walked recursively (skipping dot-folders, `target`, `dist`, `node_modules`, `AST`, `灵碑`) and treated as one program; files that do not parse are skipped with a warning.

Option	Meaning
`[path]`	File or directory (default `.`).
`--technical`, `--artwork`, `--ling`	Pick one or more drawing styles. With none of these, all three are produced.
`--all`	All three styles.
`--out <dir>`	Output folder (default `./AST`).

```
OUTPUT
$ ling ast hello.ling
ling ast: 'hello' - 1 file(s), 1 fn(s)
  ok AST\hello.technical.svg
  ok AST\hello.artwork.svg
  ok AST\hello.ling.svg
```

(The tool prints a check mark in place of `ok`.)

`ling visualize`

`ling visualize <file.ling>` writes a single SVG picture of the file's syntax tree to **standard output**; redirect it to a file.

```
OUTPUT
$ ling visualize hello.ling > hello.svg
```

`ling convert`

Transcodes an asset into an importable `.ling` file that rebuilds the asset with Ling's own builtins.

```
OUTPUT
ling convert <asset> [-o out.ling] [--no-compression]
```

Supported inputs: `.gltf` / `.glb` (meshes, node names and transforms), `.wav` / `.ogg` / `.flac` (PCM audio), `.mid` (note events), `.svg` (paths become vector strokes) and `.blend` (routed through Blender's glTF exporter when Blender is available). The output defaults to the input

name with a `.ling` extension. Bulk numeric data is embedded losslessly, deflate-compressed and base64-encoded behind the `blob_f32` / `blob_i32` builtins; `--no-compression` writes plain arrays instead. `lingfu convert` forwards to this command.

```
OUTPUT
$ ling convert s.svg -o s_out.ling
[convert] s.svg -> s_out.ling  (1 KB, deflate+base64 lossless)
```

`lingfu`

```
OUTPUT
lingfu <command> [arguments]
lingfu --help | -h | help
lingfu --version | -V | version
```

`lingfu` locates the project by searching the current directory and its parents for a manifest named `Ling.toml`, `灵符.toml`, `靈符.toml`, `영부.toml` or `ลิงฟู.toml`. The manifest's file name also decides the language of the project's folder names (below).

Commands at a glance

Command	English aliases	Purpose
`new <name>`	`create`	Create a project folder
`init`	`initialize`	Scaffold a project in the current folder
`build`	`compile`, `make`	Build a native executable
`run`	`execute`, `start`	Run the project
`test`	`verify`	Run the project's tests
`check`	`lint`, `validate`	Check sources (see warning)
`clean`	`purge`, `clear`	Delete build output
`doc`	`docs`, `document`	Build documentation
`fmt`	`format`, `style`	Format sources (see warning)
`tree`	`structure`, `ls`	Print the project tree

Command	English aliases	Purpose
`bench`	`benchmark`, `measure`	Run `cargo bench`
`update`	`upgrade`, `refresh`	Run `cargo update`
`add <package>`	`install`, `include`	Install a package from the registry
`search <query>`	`find`, `seek`	Search the registry
`publish`	`release`, `share`	Publish the project
`login <api-key>`	`sign-in`, `auth`	Save registry credentials
`logout`	`sign-out`, `deauth`	Forget credentials
`translate <file>`	`trans`, `convert`	Translate keywords between languages (see warning)
`manifest`	`show`, `info`	Print the manifest
`wizard`	`guide`, `create-wizard`	Interactive project creator
`normalize <lang>`	`norm`, `localize`	Rewrite the project into another language
`seed`	-	Integrity fingerprint of the project
`ast`, `convert`	-	Forwarded to `ling ast` / `ling convert`

Options common to several commands: `--lang` / `-l <code>` (scaffolding language: `en`, `zh`, `ja`, `ko`, `th`, `ru`), `--type` / `-t <kind>`, `--root-lang <code>`, `--crates`, `--path <dir>`, `--dry-run` / `-n`.

Localised command names. Each command also has aliases in Chinese, Japanese, Korean, Thai, Russian, Arabic, Hindi, Vietnamese, Greek, Hebrew, German, French, Spanish, Turkish and Polish (for example `试` / `测` for `test` in Chinese, `ทดสอบ` in Thai). These are recognised **only when the executable is invoked under its localised name** - `灵符.exe`, `靈符`, `영부`, `ลิงฟู`, `линфу`, `لينغفو`, `लिंगफू`, `linhphu`, `λινγφυ`, `לינגפו`, `lingfu-de`, `lingfu-fr`, `lingfu-es`, `lingfu-tr`, `lingfu-pl`. Under the normal name `lingfu` only the English words work, and messages are English:

```
OUTPUT
$ lingfu 试
unknown command: 试
  lingfu --help
$ ./灵符.exe 试          # same program, copied under its Chinese name
正在试灵...
  试验: ...\tests\test.ling
...
```

`lingfu new`

```
OUTPUT
lingfu new <name> [--type bin|lib|game|ui|ai|crypto|web|polyglot] [--lang
en|zh|ja|ko|th|ru] [--root-lang <code>]
```

Creates `<name>/` containing a manifest, a source folder with an entry program, a tests folder, a lexicon folder, `README.md`, `.gitignore` and `ling.ignore`. `--type` also accepts localised words (for example `lib` = `共修` = `라이브러리`). `--lang` chooses the language of the generated files **and** of the folder names; `--root-lang` chooses the language of the `core` dependency.

```
OUTPUT
$ lingfu new demo
~ Ling Manifest - Spirit Token Descends ~
Creating project: demo

Spirit Realm created!
  cd demo
  lingfu build
  lingfu run
$ find demo -type f
demo/.gitignore
demo/lexicons/en.ling
demo/ling.ignore
demo/Ling.toml
demo/README.md
demo/src/start.ling
demo/tests/test.ling
```

Folder and file names depend on the scaffold language:

Language	Manifest	Sources	Entry	Tests	Test file	Build	Lock	Lexicons
English	`Ling.toml`	`src`	`start.ling`	`tests`	`test.ling`	`build`	`.lock`	`lexicons`
Chinese	`灵符.toml`	`灵源`	`启.灵`	`灵镜`	`试.灵`	`灵碑`	`灵符锁`	`灵言`
Japanese	`霊符.toml`	`霊源`	`啓.霊`	`霊鏡`	`試.霊`	`霊碑`	`霊符錠`	`霊語`

Language	Manifest	Sources	Entry	Tests	Test file	Build	Lock	Lexicons
Korean	`영부.toml`	`령원`	`시작.령`	`령경`	`시험.령`	`령비`	`령잠금`	`령어`
Thai	`ลิงฟู่.toml`	`ต้น กำเนิด`	`เริ่ม.ลิง`	`ทดสอบ`	`ทดสอบ.ลิง`	`สร้าง`	`ล็อก`	`ค่าศัพท์`

(Russian, Persian, Arabic, Hebrew, Urdu, French, German and others have tables of their own.)

The generated English manifest and entry:

```toml
[spirit]
name = "demo"
kind = "bin"
version = "2030.0.0"

[author]
ling-lin = { email = "you@example.com" }

[realm]
language = "en"
level    = "铁"
source   = "src"

[root]
core = { version = "2030.0", language = "en" }

[depends]
# Add dependencies: spirit-name = { version = "x.y.z" }
```

```
FRAGMENT
# src/start.ling - gateway to the Spirit Realm

bind start = do {
    print("Welcome to demo Spirit Realm")
    print("Spirit Level: 铁")

    # Your Ling code awakens here
}
```

The `lingfu` manifest uses the tables `[spirit]` (`name`, `kind`, `version`, and optionally `entry`), `[author]`, `[realm]` (`language`, `level`, `source`), `[root]` (the `core` dependency) and `[depends]`. Two tools read it with different parsers, which matters:

- `lingfu run` parses the file as real TOML and honours `entry` in `[spirit]` (or `[package]`).
- `ling build` reads `name`, `version`, `kind`, `entry`, `icon`, `targets`, `pack`, `includes`, … line by line (see the key table above) and defaults `entry` to `main.ling`.

Because the generated project keeps its program in `src/start.ling`, **add** `entry = "src/start.ling"` **under** `[spirit]` before running `lingfu build`; without it the build stops with `entry '...\main.ling' not found`. `lingfu run` and `lingfu test` work out of the box because they search `src/start.ling` themselves.

`lingfu init`

`lingfu init [--type ...] [--lang ...]` does the same scaffolding as `new`, but inside the current directory, using its name as the project name.

`lingfu run`

Runs the project's entry program with `ling run`. The entry is the first existing file among: the manifest's `[spirit] entry`, `<src>/<entry file>`, `<root>/<entry file>`, `<src>/main.ling`, `<root>/main.ling`. If there is no manifest, it looks in the current directory for a `.ling` file (`main.ling`, `src/main.ling`, `src/start.ling`, …); as a last resort it runs `cargo run`. Extra arguments are passed to `ling run`.

```
OUTPUT
$ cd demo
$ lingfu run
Running ...\demo\src\start.ling...
Welcome to demo Spirit Realm
Spirit Level: 铁
```

`lingfu build`

Builds the project natively. If a `Cargo.toml` sits next to the manifest it simply runs `cargo build`. Otherwise it calls `ling build <project> --aot` (the manifest directory when a manifest exists, else the entry file), and passes every other option straight through: `--out`, `--`

`platform`, `--icon`, `--pack`. `--no-aot` selects the interpreter-embedding build instead of AOT compilation, and `--path <dir>` selects the project folder explicitly. The same Rust-toolchain requirement as `ling build` applies (see above). Note that `lingfu build` therefore builds the manifest's `targets` (or the host platform plus `web`), not just the host platform.

`lingfu test`

Runs `<src>/<test file>` if it exists (for English projects `src/test.ling`), then every `.ling` file below the tests folder, each with `ling run`. The first failing run (non-zero exit status) stops the sequence with `tests failed`. A project with no test files but a `Cargo.toml` falls back to `cargo test`; with no manifest at all it runs `cargo test`.

A test is an ordinary program: it needs an entry point and fails by exiting with an error. The generated test program binds `test` (any `bind` of a `do` block counts as the entry).

```
OUTPUT
$ lingfu test
Testing Ling project...
  test: ...\demo\tests\test.ling
All mirrors are clear
Ling tests passed for demo
All tests passed!
```

Because there is no assertion builtin, make a test fail deliberately with a runtime error, for example by calling `read_file` on a missing path or an unknown function under `--interp`.

`lingfu clean`

Deletes the project's build folder and lock folder (`build` and `.lock`, or their localised names) and, when a `Cargo.toml` or `target/` exists, runs `cargo clean`.

```
OUTPUT
$ lingfu clean
  removed ...\demo\build
  removed ...\demo\.lock
Clean!
```

`lingfu tree`, `lingfu manifest`

`lingfu tree` prints the project's directory tree. `lingfu manifest` prints the path of the manifest followed by its contents (or the `Cargo.toml` if no Ling manifest exists).

```
OUTPUT
$ lingfu tree
Project tree:
demo
├── .gitignore
├── Ling.toml
├── README.md
├── lexicons
├   └── en.ling
├── ling.ignore
├── src
├   └── start.ling
└── tests
└   └── test.ling
```

The branch drawing is the tool's own: it prefixes every child with the parent's connector, so nested entries show two connectors on one line.

`lingfu normalize`

```
OUTPUT
lingfu normalize <lang> [--dry-run|-n] [--content-only] [--files-only]
lingfu normalize --lang <lang>
```

Rewrites the whole project (the current directory) into one human language: it translates the keywords and builtin names inside every source file, renames source files and the standard folders to the target language's conventions, and rewrites imports to match. `<lang>` is a language code or name: `en`, `zh`, `ja`, `ko`, `th`, `fa`, `ar`, `he`, `ur`, `fr`, `de`, … (full names such as `chinese`, native names such as `中文`, and other spellings also work); without it the command shows an interactive menu offering English, Chinese, Japanese, Korean and Thai. `--dry-run` lists what would change without touching anything; `--content-only` rewrites file contents but not names; `--files-only` renames but does not rewrite contents.

```
OUTPUT
$ lingfu normalize zh --dry-run
Normalizing to Chinese  [dry-run]
  ~ lexicons\en.ling
  ~ src\start.ling
  ~ tests\test.ling
  mv tests\test.ling -> tests\测试.ling
  mv build -> 构建
  mv src -> 源码
  mv tests -> 测试

Dry run complete - no files changed.
  3 file(s) rewritten
  1 file(s) renamed
  3 folder(s) renamed
```

Note that `normalize` uses its own folder naming (`源码`, `测试`, `构建`), which differs from the names `lingfu new --lang zh` creates (`灵源`, `灵镜`, `灵碑`). Run a dry run first and commit the project before applying it.

Registry commands: `login`, `logout`, `add`, `search`, `publish`

The package registry defaults to `https://fu.ling-lang.org`; set the environment variable `LINGFU_REGISTRY` to use another. The registry client is not exercised by the examples in this book (they need a network and an account).

Command	Behaviour
`login <api-key>`	Saves the key to `~/.lingfu/credentials.toml` (and the registry URL). Create keys in the registry's developer portal.
`logout`	Removes the saved credentials.
`publish`	Packs the project as a `.tar.gz`, skipping files listed in `ling.ignore` (one glob per line, e.g. `*.key`, `*.pem`, `build/`), and uploads it as a streamed multipart request with a progress bar. Requires `login`.
`search <query>`	Searches the registry. `--crates` searches crates.io through `cargo search` instead.
`add <package>`	Downloads a package and unpacks it into `./ling_modules/<name>/`, then prints the file to import: `use "ling_modules/<name>/<entry>";`. `--crates` shells out to `cargo add` for Rust dependencies.

`lingfu wizard`

An interactive questionnaire (language, project type, name, …) that ends by scaffolding a project like `new`. It reads from the terminal and therefore cannot be scripted.

Quick reference: project layout and daily workflow

```
OUTPUT
$ lingfu new shop --type game
$ cd shop
$ lingfu run              # run src/start.ling
$ lingfu test             # run tests/*.ling
$ ling run --interp src/start.ling     # run one file with full diagnostics
$ ling build . --platform win,web --out dist
$ lingfu normalize ja --dry-run
```

Exit status: both tools return 0 on success and 1 on any reported failure (`lingfu` prints `Error: <message>` for failures such as `build failed`, `tests failed`, `run failed`).

CHAPTER 20

Window, Frame Loop, Input and Camera

This chapter covers everything a graphical Ling program needs before it draws a single shape: opening a window, running the frame loop, painting in two dimensions, reading the keyboard, mouse and gamepad, and aiming the 3D camera. Shapes, textures and effects are in the next chapter.

Every entry below was checked against the interpreter's dispatch table (`src/runtime/mod.rs`). Where this book and the older draft pages in the repository disagree, this book follows the source. Entries marked "Aliases" list the Chinese (zh), Japanese (ja), Korean (ko) and Thai (th) names accepted by the interpreter in addition to the English name; the interpreter also accepts Arabic, Hebrew, Persian, Urdu, French, German and Russian names for many builtins, which are not listed here.

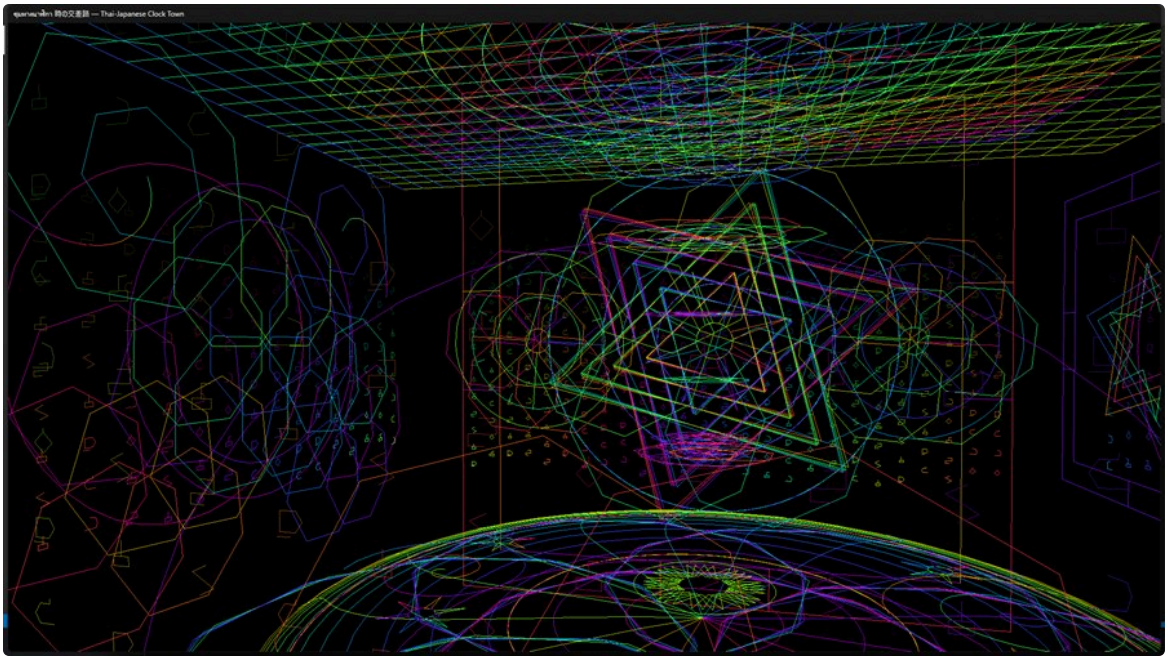


Figure 10.1. A scene built only from the builtins in this chapter and the next, drawn live in a Ling window. Source: `examples/3d-rooms/3d-thai-japanese-clocktown.ling` *in the Ling repository.*

The frame loop

A Ling graphics program has one shape. Open a window, then repeat four steps until the window closes: clear, draw, `present`, repeat.

```
GUI
bind start = do {
    open_window(800, 600, "Frame loop")
    bind frame = 0
    while window_is_open() {
        fill(10, 12, 30)                    # 1. clear to a background colour
        set_color(255, 200, 80)             # 2. draw
        draw_disc(400 + 200 * sin(frame * 0.05), 300, 40)
        present()                           # 3. flush and show
        bind frame = frame + 1
    }
}
```

Three facts about this loop are worth memorising.

1. `window_is_open()` becomes false when the user closes the window **or presses Escape**. You do not need to handle Escape yourself.
2. `present()` is the only call that puts pixels on screen. It also flushes everything queued by 3D builtins, runs the post-processing chain, and reads the mouse. A frame without `present()` is invisible.
3. There is no delta-time builtin. Use `time_now()` (seconds since the program started) and subtract the previous value, or use `frame_count()`.

Colour arguments are plain numbers from 0 to 255. They are **not clamped**: the interpreter packs `(r << 16) | (g << 8) | b`, so a value above 255 spills into the neighbouring channel. Negative numbers become 0.

Calling a function that does not exist (for example `draw_rect`, which Ling does not have) is silently ignored. If a drawing call seems to do nothing, check the spelling against this chapter first.

Coordinate conventions

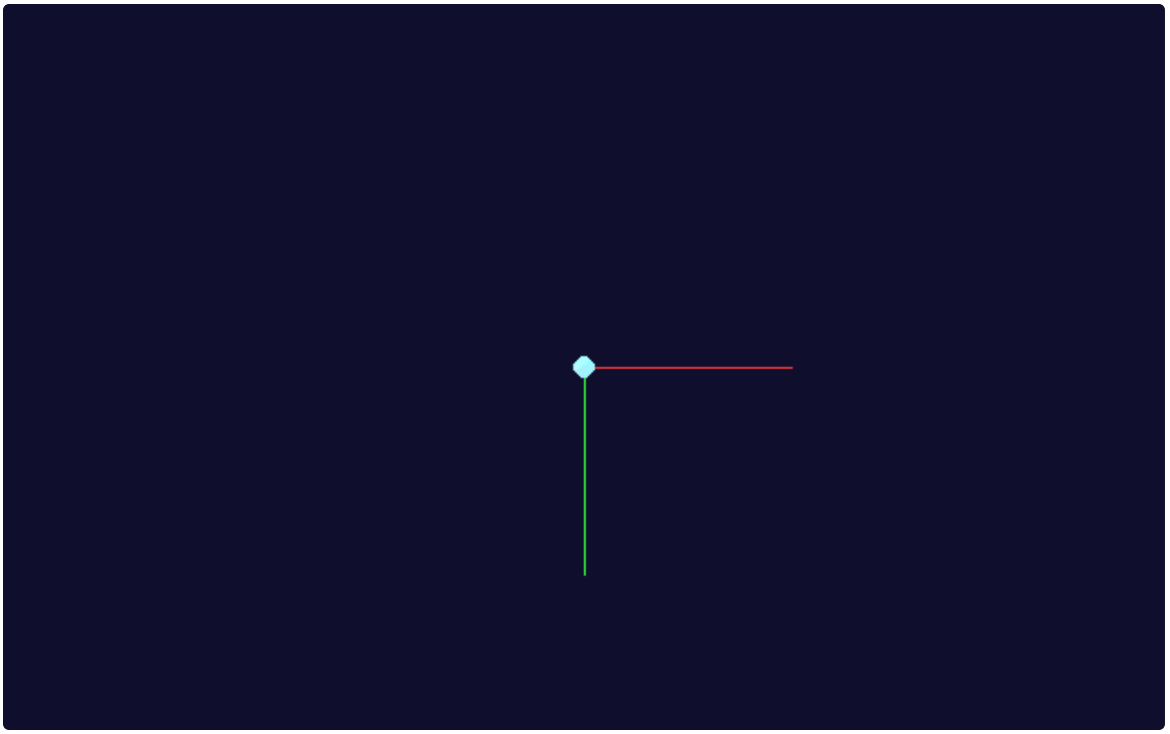


Figure 10.2. A red line along +x, a green line along +y and a blue line along +z, all starting at the white sphere. The green line points down the screen. The blue line points straight into the screen, so it is seen end-on.

Space	Origin	+x	+y	+z
Screen (2D)	top-left pixel	right	**down**	not used
World (3D)	camera pivot	right	**down**	away from the camera

Because +y is down, "above" is negative y. A ceiling lives at `y = -3`, a floor at `y = +3`. Angles in `draw_arc` also run clockwise on the screen for the same reason.

The default projection centres the view on the window, uses a focal length equal to the window height in pixels, and adds a fixed depth offset `zdist = 5` before the perspective divide. A point at `(x, y, z)` with the camera at the origin lands at

```
screen_x = width/2  + height * x / (z + zdist)
screen_y = height/2 + height * y / (z + zdist)
```

Summary: window and frame

Builtin	Purpose
`open_window(w, h, title, resizable)`	Open a window
`open_fullscreen(title, w, h)`	Open a borderless full-screen window
`open_fullscreen_scaled(title, scale)`	Full screen at 1/scale resolution
`window_is_open()`	Loop condition
`present()`	Finish the frame
`quit()`	Close the window from script
`get_width()`, `get_height()`	Framebuffer size
`set_fps(n)`, `set_vsync(on)`	Frame pacing
`monitor_width()`, `monitor_height()`, `monitor_refresh()`, `monitor_info()`	Physical display
`flush_3d()`, `flush_post()`	Mid-frame flush
`wait_window()`	Keep the last frame on screen
`screenshot(tag)`, `record_frame()`	Save the framebuffer
`frame_count()`, `time_now()`	Clocks

Window and frame entries

`open_window(w, h, title, resizable)`

Opens the program's window and allocates its framebuffer.

- **Aliases:** en `gfx_window`; zh `开窗`; ja `ウィンドウ開く`; ko `창열기`; th `เปิดหน้าต่าง`
- **Parameters:** `w` width in pixels (default 800); `h` height in pixels (default 600); `title` window title string (default `"Ling"`); `resizable` optional, `1` makes the window resizable and maximisable (default 0).
- **Returns:** nothing.
- **Notes:** The numeric arguments come first and the title third. When the window is resizable the framebuffer follows the client size from the next frame on, so read `get_width()` and `get_height()` each frame. Opening the window also resets the 3D projection to its defaults (centre of window, focal length equal to the height, `zdist` 5). If the window cannot be created the program stops with `cannot open window`.

```
GUI
open_window(1024, 640, "My game", 1)
```

`open_fullscreen(title, w, h)`

Opens a borderless window covering the whole primary monitor.

- **Aliases:** en `fullscreen`; zh `全屏`; ja `全画面`; ko `전체화면`; th `เปิดหน้าต่างเต็มจอ`
- **Parameters:** `title` window title (default `"Ling"`); `w`, `h` optional pixel size (default: the monitor's resolution).
- **Returns:** nothing.
- **Notes:** The title is the **first** argument here, unlike `open_window`. On Windows the window is topmost and keeps keyboard focus; `examples/geometry/shapes_demo.ling` opens its window this way and pairs it with `capture_mouse()`.

```
GUI
open_fullscreen("Cathedral")
```

`window_is_open()`

True while the window is open; false once it has been closed, `quit()` was called, or Escape was pressed.

- **Aliases:** en `gfx_is_open`, `is_open`; zh `窗开`; ja `開いている`; ko `창열림`; th `หน้าต่างเปิดอยู่`
- **Parameters:** none.
- **Returns:** boolean.
- **Notes:** Before any window is opened it returns false, so a loop guarded by it simply does not run.

```
GUI
while window_is_open() {
    fill(0, 0, 0)
    present()
}
```

`present()`

Ends the frame: draws every queued 3D primitive, applies post-processing, copies the framebuffer to the window and samples the mouse.

- **Aliases:** en `gfx_present`, `show`; zh `呈现`, `显`; ja `表示`; ko `표시`; th `แสดงผล`
- **Parameters:** none.
- **Returns:** nothing.
- **Notes:** Depth-sorted 3D geometry (`cube`, `vtex_*`, `draw_line_3d`, and the others in the next chapter) is only rasterised here, so those calls can be made in any order within the frame. Direct 2D calls (`draw_line`, `triangle`, `draw_disc`, `tex_*`) write to the framebuffer immediately, before the queue is flushed, so 3D geometry always lands on top of them. To draw 2D interface over a 3D scene use `flush_3d()` first. Click detection for the `ui_*` widgets depends on `present()` being called once per frame.

```
GUI
present()
```

`quit()`

Asks the window to close, exactly like pressing Escape. Useful for an on-screen exit button.

- **Aliases:** English only; `exit_game` and `close_window` are accepted synonyms.
- **Parameters:** none.
- **Returns:** nothing.

```
GUI
if ui_button(20, 20, 100, 32) == 1 { quit() }
```

`get_width()` and `get_height()`

The current framebuffer width and height in pixels.

- **Aliases, `get_width`:** zh `宽`; ja `幅取得`; ko `너비`; th `ความกว้าง`
- **Aliases, `get_height`:** zh `高`; ja `高取得`; ko `높이`; th `ความสูง`
- **Parameters:** none.
- **Returns:** number.

- **Notes:** For a fixed-size window these equal the arguments of `open_window`. For a resizable window they change when the user resizes it.

```
GUI
bind cx = get_width() / 2
bind cy = get_height() / 2
```

`set_fps(n)`

Caps the frame rate.

- **Aliases:** en `set_target_fps`, `target_fps`; zh `设帧率`; ja `フレームレート設定`; ko `프레임설정`; th `ตั้งเฟรมเรต`
- **Parameters:** `n` frames per second (default 60, minimum 1).
- **Returns:** nothing.
- **Notes:** Takes effect on the open window, so call it after `open_window`.

```
GUI
open_window(800, 600, "Capped")
set_fps(30)
```

`monitor_width()`, `monitor_height()`, `monitor_refresh()`

The primary monitor's pixel width, pixel height and refresh rate in hertz.

- **Aliases, `monitor_width`:** en `screen_width`; zh `屏宽`; ja `画面幅`; ko `화면너비`; th `ความกว้างจอ`
- **Aliases, `monitor_height`:** en `screen_height`; zh `屏高`; ja `画面高`; ko `화면높이`; th `ความสูงจอ`
- **Aliases, `monitor_refresh`:** en `monitor_hz`, `monitor_fps`, `refresh_rate`; zh `刷新率`; ja `リフレッシュレート`; ko `주사율`; th `อัตรารีเฟรช`
- **Parameters:** none.
- **Returns:** number.

```
bind start = do {
    println("monitor: " + str(monitor_info()))
}
```

```
OUTPUT
monitor: [1920, 1080, 60]
```

The values depend on the machine that runs the example; this output came from a 1920 by 1080 display at 60 Hz.

`monitor_info()`

The three monitor values as a list `[width, height, refresh_hz]`.

- **Aliases:** en `screen_info`; zh `屏幕信息`; ja `画面情報`; ko `화면정보`; th `ข้อมูลจอ`
- **Parameters:** none.
- **Returns:** list of three numbers.

```
GUI
bind info = monitor_info()
open_window(info[0] / 2, info[1] / 2, "Half-screen")
```

`wait_window()`

Blocks, keeping the last framebuffer on screen, until the window is closed or Escape is pressed.

- **Aliases:** en `gfx_wait`; th `รอหน้าต่าง`
- **Parameters:** none.
- **Returns:** nothing.
- **Notes:** Handy for programs that draw one still image and then wait.

```
GUI
open_window(400, 300, "Still")
fill(20, 40, 80)
wait_window()
```

`screenshot(tag)`

Saves the current framebuffer as a PNG and returns its path.

- **Aliases:** th `บันทึกภาพ`
- **Parameters:** `tag` a short label baked into the file name (default `"game"`); non-alphanumeric characters become underscores.
- **Returns:** string, the relative path `screenshots/ss_<unix-time>_<tag>_<width>x<height>.png`.
- **Notes:** The folder `screenshots` is created next to where the program is run. The image is the framebuffer as of the last `present()`, so call it after `present()` for a complete frame. Every screenshot in this book was taken this way.

```
GUI
present()
println(screenshot("title-screen"))
```

`frame_count()`

How many frames have been presented since the program started.

- **Aliases:** zh `帧数`; ja `フレーム数`; ko `프레임수`; th `เฟรม`
- **Parameters:** none.
- **Returns:** number.

```
GUI
if frame_count() % 60 == 0 { println("one more second") }
```

`time_now()`

Seconds elapsed since the program started, as a fractional number.

- **Aliases:** zh `当前时间`; ja `経過時間`; ko `현재시간`; th `เวลาปัจจุบัน`
- **Parameters:** none.
- **Returns:** number.
- **Notes:** The idiom for a frame-time delta is to remember the previous value. (`epoch_now()` returns wall-clock seconds since 1970.)

```
GUI
bind last = time_now()
while window_is_open() {
    bind now = time_now()
    bind dt = now - last
    bind last = now
    fill(0, 0, 0)
    present()
}
```

2D drawing

All the 2D builtins use screen pixels with the origin at the top-left and +y down. They draw in the current **pen colour**, which stays in effect until you change it. `present()` is not needed for the pixels to be written, only for them to be shown.

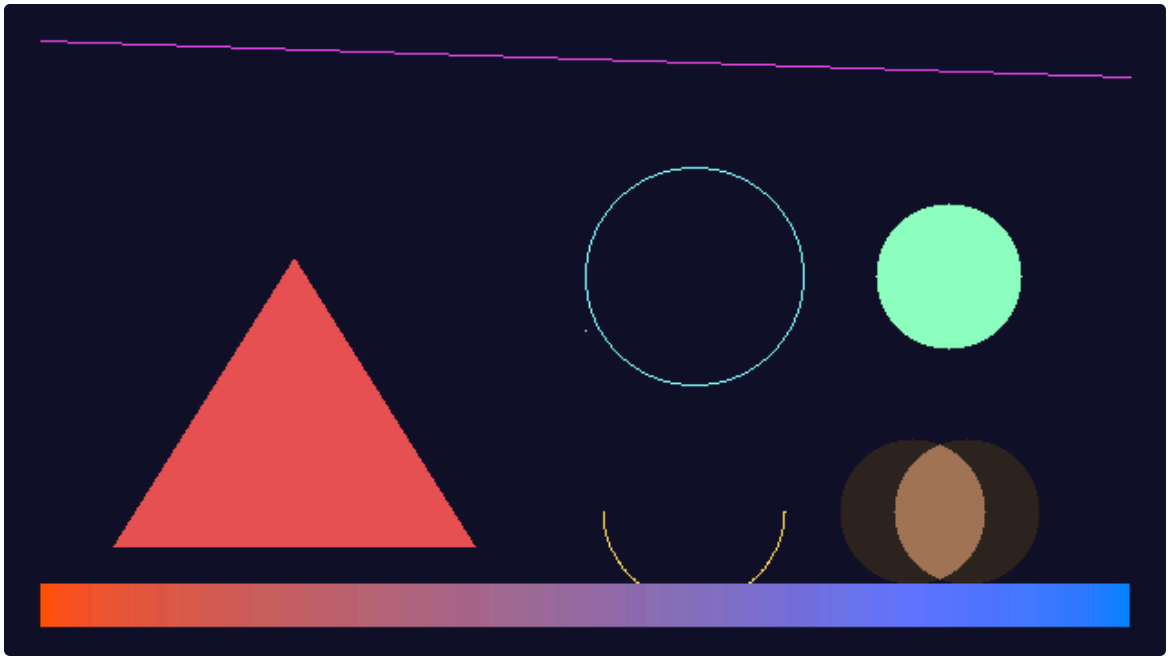


Figure 10.3. One frame of 2D drawing. The brown circles at the lower right overlap under `set_blend(1)`*, which adds colour, so the overlap is brighter. The yellow arc was drawn from angle 0 to pi and, because y points down, it is the lower half of its circle.*

Builtin	Purpose
`fill(r, g, b)`	Clear the framebuffer
`set_color(r, g, b)`, `set_color_hsl(h, s, l)`, `mix_color(...)`	Pen colour
`set_alpha(a)`, `set_blend(mode)`, `set_antialias(on)`	Pen state
`draw_pixel`, `draw_line`, `triangle`	Basic primitives
`draw_circle`, `draw_disc`, `draw_arc`, `draw_poly`	Curves and polygons
`grad_rect`, `grad_triangle`	Gradient fills
`shadow_blob`, `cast_shadow`, `shadow_params`	Soft shadows
`depth_triangle`, `depth_line`	Depth-sorted 2D
`set_frame_blur(a)`	Afterimage trails

`fill(r, g, b)`

Clears the whole framebuffer to a colour and resets the depth buffer for the new frame.

- **Aliases:** en `gfx_fill`, `clear`; zh `填`, `清`; ja `塗り潰し`, `消去`; ko `지우기`, `채우기`; th `เติม`
- **Parameters:** `r`, `g`, `b` each 0 to 255 (each defaults to 0, so `fill()` clears to black).
- **Returns:** nothing.
- **Notes:** Call it first in every frame. It does not change the pen colour.

```
GUI
fill(10, 12, 30)
```

`set_color(r, g, b)`

Sets the pen colour used by every drawing call that follows. 3D shapes also use it as their base colour.

- **Aliases:** en `gfx_color`, `color`; zh `设色`; ja `色設定`; ko `색설정`; th `สีดินสอ`
- **Parameters:** `r`, `g`, `b` each 0 to 255 (default 255).
- **Returns:** nothing.

```
GUI
set_color(255, 120, 60)
draw_disc(100, 100, 30)
```

`set_color_hsl(h, s, l)`

Sets the pen colour from hue, saturation and lightness.

- **Aliases:** zh `色相` , `颜色HSL` ; ja `HSL色` ; ko `HSL색설정` ; th `สีHSLวาด`
- **Parameters:** `h` hue in degrees, 0 to 360 (default 0); `s` saturation 0 to 100 (default 70); `l` lightness 0 to 100 (default 50).
- **Returns:** nothing.
- **Notes:** Convenient for rainbows: step `h` by a small amount per frame or per object.

```
GUI
bind i = 0
while i < 12 {
    set_color_hsl(i * 30, 80, 55)
    draw_disc(60 + i * 50, 200, 20)
    bind i = i + 1
}
```

`set_alpha(a)`

Pen opacity for the gradient fills (`grad_rect` , `grad_triangle`) and for queued 3D draws.

- **Aliases:** zh `设透明` ; ja `アルファ設定` ; ko `투명도설정` ; th `ตั้งความโปร่งใส`
- **Parameters:** `a` from 0 (invisible) to 1 (opaque), clamped (default 1).
- **Returns:** nothing.

```
GUI
set_alpha(0.4)
```

`set_blend(mode)`

Chooses how new pixels combine with what is already there.

- **Aliases:** zh `混合模式` ; ja `ブレンドモード` ; ko `블렌드모드` ; th `โหมดผสม`
- **Parameters:** `mode` an integer. `0` normal (default), `1` additive. The queued 3D path also recognises `2` multiply, `3` screen, `4` subtract and `5` overlay (this list is from the comment in `src/gfx/depth.rs` ; the immediate 2D calls distinguish only 0 and 1).
- **Returns:** nothing.

- **Notes:** Additive blending is the basis of every glow effect: overlapping shapes get brighter. Remember to set it back to 0.

```
GUI
set_blend(1)
set_color(60, 40, 20)
draw_disc(500, 280, 40)
draw_disc(530, 280, 40)       # the overlap is brighter
set_blend(0)
```

`draw_pixel(x, y)`

Sets one pixel to the pen colour. Coordinates outside the window are ignored.

- **Aliases:** en `gfx_pixel`, `pixel`; zh `画点`; ja `点描く`; ko `점그리기`; th `วาดจุด`
- **Parameters:** `x`, `y` pixel coordinates (truncated to integers).
- **Returns:** nothing.

```
GUI
draw_pixel(320, 180)
```

`draw_line(x0, y0, x1, y1)`

A one-pixel line in the pen colour.

- **Aliases:** en `gfx_line`, `line`; zh `画线`; ja `線描く`; ko `선그리기`; th `วาดเส้น`
- **Parameters:** the two end points in pixels.
- **Returns:** nothing.
- **Notes:** Honours `set_antialias` and additive blend.

```
GUI
set_color(255, 255, 255)
draw_line(0, 0, get_width(), get_height())
```

`triangle(x0, y0, x1, y1, x2, y2)`

A filled triangle in the pen colour.

- **Aliases:** en `draw_triangle`, `gfx_triangle`; zh `三角形描画`, `画三角`; ko `삼각형그리기`; th `วาดสามเหลี่ยม`
- **Parameters:** three corner points in pixels.
- **Returns:** nothing.
- **Notes:** To build rectangles and other polygons from triangles use two or more calls, or use `draw_poly`.

```
GUI
set_color(230, 90, 90)
triangle(60, 300, 160, 140, 260, 300)
```

`draw_circle(cx, cy, r)`

A circle outline.

- **Aliases:** zh `円描画`, `画圆`; ko `원그리기`; th `วาดวงกลม`
- **Parameters:** centre `cx`, `cy` and radius `r` in pixels (radius default 10).
- **Returns:** nothing.

```
GUI
set_color(90, 200, 255)
draw_circle(380, 150, 60)
```

`draw_disc(cx, cy, r)`

A filled circle.

- **Aliases:** en `draw_filled_circle`; zh `画实心圆`; ja `塗りつぶし円`; ko `원채우기`; th `วาดวงกลมทึบ`
- **Parameters:** centre and radius as for `draw_circle`.
- **Returns:** nothing.
- **Notes:** Honours the blend mode, so with `set_blend(1)` it paints a glow.

```
GUI
set_color(120, 255, 160)
draw_disc(520, 150, 40)
```

`draw_arc(cx, cy, r, a0, a1, segments)`

Strokes part of a circle.

- **Aliases:** en `arc` ; zh `円弧描画` , `画弧` ; ko `호그리기` ; th `วาดส่วนโค้ง`
- **Parameters:** centre and radius; `a0` and `a1` start and end angles in radians (defaults 0 and 2 pi, a full circle); `segments` optional number of line segments.
- **Returns:** nothing.
- **Notes:** Angles grow clockwise on screen because +y is down: angle 0 is to the right, `pi / 2` is straight down.

```
GUI
set_color(255, 220, 90)
draw_arc(380, 280, 50, 0, 3.14159)     # the lower half of the circle
```

`draw_poly(points)`

A filled 2D polygon that closes itself.

- **Aliases:** zh `填充多边形` ; ja `ポリゴン塗り` ; ko `다각형채우기` ; th `เติมรูปหลายเหลี่ยม`
- **Parameters:** `points` a flat list `[x0, y0, x1, y1, ...]` .
- **Returns:** nothing.
- **Notes:** Honours blend mode. Pair it with `project_3d` to put a filled shape on projected 3D points.

```
GUI
set_color(200, 140, 255)
draw_poly([100, 100, 200, 120, 180, 200, 90, 180])
```

`grad_rect(x, y, w, h, r0, g0, b0, r1, g1, b1, dir)`

A rectangle filled with a linear gradient between two colours.

- **Aliases:** zh `渐变矩形` ; ja `グラデ矩形` ; ko `그라데사각` ; th `สี่เหลี่ยมไล่สี`
- **Parameters:** position and size in pixels; the start colour then the end colour as 0 to 255 triples; `dir` `0` for left to right, anything else for top to bottom (default 1).
- **Returns:** nothing.

- **Notes:** Honours `set_alpha`. Interpolation is perceptual by default; `set_gradient_space(0)` switches to plain sRGB.

```
GUI
grad_rect(20, 320, 600, 24,  255, 80, 0,  0, 80, 255,  0)
```

`key_down(name)`

Is the named key currently held?

- **Aliases:** zh `按键`; ja `キー押す`; ko `키누름`; th `กดค้าง`
- **Parameters:** `name` a lower-case string (see the table below).
- **Returns:** boolean. An unknown name is simply false.
- **Notes:** Returns false while another window has focus (and for a short grace period after focus returns).

Group	Names
Letters, digits	`"a"` to `"z"`, `"0"` to `"9"`
Arrows and navigation	`"left"`, `"right"`, `"up"`, `"down"`, `"home"`, `"end"`, `"pageup"`, `"pagedown"`, `"insert"`, `"delete"`
Editing	`"space"`, `"enter"`, `"escape"`, `"tab"`, `"backspace"`, `"capslock"`
Modifiers	`"lshift"`, `"rshift"`, `"lctrl"`, `"rctrl"`, `"lalt"`, `"ralt"`
Function keys	`"f1"` to `"f12"`
Number pad	`"numpad0"` to `"numpad9"` (or `"kp0"` to `"kp9"`), `"numpad+"`, `"numpad-"`, `"numpad*"`, `"numpad/"`
Punctuation	`"minus"`, `"equals"`, `"comma"`, `"period"`, `"slash"`, `"semicolon"`, `"quote"`, `"lbracket"`, `"rbracket"`, `"backslash"`, `"grave"` (the symbols `-` `=` `,` `.` `/` `;` `'` `[` `]` `` ` `` also work)

```
GUI
bind x = 400
while window_is_open() {
    if key_down("left")  { bind x = x - 4 }
    if key_down("right") { bind x = x + 4 }
    fill(0, 0, 0)
    set_color(255, 255, 255)
    draw_disc(x, 300, 20)
    present()
}
```

`key_pressed(name)`

True only on the single frame in which the key went down, so holding the key does not repeat.

- **Aliases:** zh `键按` ; ja `キー押した` ; ko `키눌림` ; th `กดปุ่ม`
- **Parameters:** `name` as for `key_down` .
- **Returns:** boolean.
- **Notes:** Use it for toggles, menu selection and firing once per press. Pressing Start on a gamepad counts as `"enter"` .

```
GUI
bind paused = 0
while window_is_open() {
    if key_pressed("p") { bind paused = 1 - paused }
    fill(0, 0, 0)
    present()
}
```

`mouse_x()` and `mouse_y()`

The cursor position inside the window, in pixels, clamped to the window.

- **Aliases:** none (English names only)
- **Parameters:** none.
- **Returns:** number.

- **Notes:** Both return 0 until the window exists. `ui_hot` and the `ui_*` controls read the mouse themselves, so you rarely need these for interface work.

```
GUI
set_color(255, 255, 255)
draw_disc(mouse_x(), mouse_y(), 6)
```

`mouse_down()`, `mouse_down_right()`, `mouse_down_middle()`

True while the left, right or middle button is held.

- **Aliases,** `mouse_down_right`**:** th `เมาส์ขวา`
- **Aliases,** `mouse_down_middle`**:** th `เมาส์กลาง`
- **Parameters:** none.
- **Returns:** boolean.
- **Notes:** These are level checks. For a click edge, remember last frame's value.

```
GUI
bind was = 0
while window_is_open() {
    bind now = 0
    if mouse_down() { bind now = 1 }
    if now == 1 and was == 0 { println("click at " + str(mouse_x()) + "," +
str(mouse_y())) }
    bind was = now
    fill(0, 0, 0)
    present()
}
```

`mouse_dx()` and `mouse_dy()`

How far the mouse moved since the previous frame, in pixels. Positive `dx` is rightward and positive `dy` is downward.

- **Aliases,** `mouse_dx`**:** zh `鼠ΔX`; ja `マウスΔX`; ko `마우스ΔX`; th `เมาส์X`
- **Aliases,** `mouse_dy`**:** zh `鼠ΔY`; ja `マウスΔY`; ko `마우스ΔY`; th `เมาส์Y`
- **Parameters:** none.
- **Returns:** number.

- **Notes:** These are only non-zero while the mouse is captured (`capture_mouse()`); they are the basis of first-person look. When captured on Windows the cursor is re-centred every frame, so turning never stops at a screen edge.

```
GUI
capture_mouse()
bind yaw = 0.0
while window_is_open() {
    bind yaw = yaw + mouse_dx() * 0.002
    fill(0, 0, 0)
    present()
}
```

`capture_mouse()` and `release_mouse()`

Hide the cursor and switch to relative motion, so that `mouse_dx()` and `mouse_dy()` keep reporting movement however far the mouse travels; `release_mouse()` undoes it.

- **Aliases:** zh `捕鼠` ; ja `マウス捕捉` ; ko `마우스잡기` ; th `จับเมาส์`
- **Parameters:** none.
- **Returns:** nothing.
- **Notes:** Call `capture_mouse()` once, after the window is open. Escape still closes the window.

```
GUI
open_fullscreen("Look around")
capture_mouse()
```

`text_poll()`, `text_get()`, `text_set(s)`, `text_clear()`

A typed-text buffer for text fields.

- **Aliases:** English names only.
- **Parameters:** `text_set(s)` takes the new contents as a string; the others take none.
- **Returns:** `text_poll()` folds the keys typed since the last call into the buffer (with key repeat) and returns the whole buffer as a string; `text_get()` returns it without polling; `text_set` and `text_clear` return nothing.

- **Notes:** The buffer has a length cap. Backspace edits it. Use `font_text_fill` or `ui_text` to display it.

```
GUI
bind f = font_load("assets/fonts/Exo2.ttf", 600)
while window_is_open() {
    bind s = text_poll()
    fill(10, 14, 24)
    set_color(255, 255, 255)
    font_text_fill(f, 40, 80, 28, "> " + s)
    present()
}
```

`pad_poll()`

Reads the gamepads for this frame. Call it once per frame before the other `pad_*` functions.

- **Aliases:** zh `手柄轮询` ; ja `パッド更新` ; ko `패드폴링` ; th `อัปเดตแพด`
- **Parameters:** none.
- **Returns:** the number of connected pads (0 if none or if the input system cannot start).
- **Notes:** The player slot argument `i` in the other `pad_*` builtins starts at 0.

```
GUI
bind n = pad_poll()
```

`pad_count()` **and** `pad_connected(i)`

How many pads are connected, and whether the pad in slot `i` is.

- **Aliases,** `pad_count` **:** zh `手柄数` ; ja `パッド数` ; ko `패드수` ; th `จำนวนแพด`
- **Aliases,** `pad_connected` **:** zh `手柄连接` ; ja `パッド接続` ; ko `패드연결` ; th `แพดเชื่อม`
- **Parameters:** `pad_connected(i)` slot index.
- **Returns:** number, and boolean.

```
GUI
if pad_connected(0) { println("player one has a pad") }
```

`pad_button(i, name)` and `pad_pressed(i, name)`

Is the button held, and did it go down this frame.

- **Aliases:** zh `手柄按键`; ja `パッドボタン`; ko `패드버튼`; th `ปุ่มแพด`
- **Parameters:** `i` slot; `name` a button name: `"a"`, `"b"`, `"x"`, `"y"` (also `"south"`, `"east"`, `"west"`, `"north"`, `"cross"`, `"circle"`, `"square"`, `"triangle"`), `"lb"` / `"l1"`, `"rb"` / `"r1"`, `"lt"` / `"l2"`, `"rt"` / `"r2"`, `"start"`, `"select"`, `"guide"`, `"l3"`, `"r3"`, `"up"`, `"down"`, `"left"`, `"right"` (the last four are the d-pad).
- **Returns:** boolean.

```
GUI
pad_poll()
if pad_pressed(0, "a") { println("jump") }
```

`pad_lx(i)`, `pad_ly(i)`, `pad_rx(i)`, `pad_ry(i)`, `pad_lt(i)`, `pad_rt(i)`

Analogue stick axes (about -1 to 1) and trigger pressure (0 to 1) for the pad in slot `i`.

- **Aliases, `pad_lx`:** zh `手柄左X`; ja `パッド左X`; ko `패드왼X`; th `แพดซ้ายX`
- **Aliases, `pad_ly`:** zh `手柄左Y`; ja `パッド左Y`; ko `패드왼Y`; th `แพดซ้ายY`
- **Aliases, `pad_rx`:** zh `手柄右X`; ja `パッド右X`; ko `패드오X`; th `แพดขวาX`
- **Aliases, `pad_ry`:** zh `手柄右Y`; ja `パッド右Y`; ko `패드오Y`; th `แพดขวาY`
- **Aliases, `pad_lt`:** zh `手柄左扳机`; ja `パッド左トリガー`; ko `패드왼트리거`; th `ไกแพดซ้าย`
- **Aliases, `pad_rt`:** zh `手柄右扳机`; ja `パッド右トリガー`; ko `패드오트리거`; th `ไกแพดขวา`
- **Parameters:** `i` slot index.
- **Returns:** number. Zero when no pad is present.

```
GUI
pad_poll()
bind x = x + pad_lx(0) * 4
```

`set_camera(cry, sry, crx, srx)`

Sets the camera orientation from precomputed cosines and sines.

- **Aliases:** zh `设置摄像机`, `设镜`; ja `カメラ設定`; ko `카메라설정`; th `ตั้งกล้อง`

- **Parameters:** `cry`, `sry` cosine and sine of the yaw angle (defaults 1 and 0); `crx`, `srx` cosine and sine of the pitch angle (defaults 1 and 0).
- **Returns:** nothing.
- **Notes:** Pass the four numbers, not the angles. `set_camera(1, 0, 1, 0)` is the neutral view straight along +z.

```
bind start = do {
    bind a = 0.3
    set_camera(cos(a), sin(a), 1, 0)
    println("yaw+  : " + str(project_3d(0, 0, 5)))
    set_camera(1, 0, cos(a), sin(a))
    println("pitch+: " + str(project_3d(0, 0, 5)))
}
```

```
OUTPUT
yaw+  : [796.77392578125, 540, 4.776682376861572]
pitch+: [960, 376.7739562988281, 4.776682376861572]
```

With no window open the projection keeps its defaults for a 1920 by 1080 frame (centre 960, 540). A positive yaw moved the point left of centre; a positive pitch moved it up.

`set_camera_pos(x, y, z)`

Moves the camera pivot to a world position.

- **Aliases:** zh `镜坐标`; ja `カメラ座標`; ko `카메라좌표`; th `ตั้งตำแหน่งกล้อง`
- **Parameters:** `x`, `y`, `z` in world units (default 0).
- **Returns:** nothing.
- **Notes:** The audio listener follows the camera position when audio is active. Remember y is down: to raise the camera, decrease y.

```
GUI
set_camera_pos(0, -2, -6)       # two units up, six back
```

`set_zdist(d)`

Sets the depth offset added before the perspective divide. A larger value moves the whole scene away and flattens perspective; a smaller one exaggerates it.

- **Aliases:** zh `镜距`; ja `Z距離設定`; ko `Z거리설정`; th `ตั้งระยะห่าง`
- **Parameters:** `d` (default 5). The window sets 5 when it opens.
- **Returns:** nothing.
- **Notes:** The repository's draft page calls this the "near-plane distance"; in the source it is a depth offset. Geometry nearer than `-zdist` is clipped away.

```
GUI
set_zdist(8)
```

`project_3d(x, y, z)`

Projects a world point to the screen with the current camera.

- **Aliases:** zh `投影3D`; ja `3D投影`; ko `3D투영`; th `ฉาย3มิติ`
- **Parameters:** a world position.
- **Returns:** a list `[screen_x, screen_y, depth]`. If the point is behind the camera's near plane the list is `[-99999, -99999, depth]`.
- **Notes:** Use it to attach 2D labels to 3D objects.

```
bind start = do {
    set_camera(1, 0, 1, 0)
    set_camera_pos(2, 0, 0)
    println("camera x+2: " + str(project_3d(0, 0, 5)))
    set_camera_pos(0, 0, 0)
    set_zdist(10)
    println("zdist 10  : " + str(project_3d(0, 0, 5)))
    println("behind    : " + str(project_3d(0, 0, -20)))
}
```

```
OUTPUT
camera x+2: [744, 540, 5]
zdist 10  : [960, 540, 5]
behind    : [-99999, -99999, -20]
```

Moving the camera two units right moved the point 216 pixels left (a lateral offset of 2 units at depth 10 with focal length 1080).

A free-fly camera

The whole chapter in one program: mouse look, WASD walking and a spinning object to look at. The forward vector `(sin yaw, cos yaw)` keeps the walking direction aligned with the view.

GUI

```
bind start = do {
    open_window(900, 600, "Fly")
    capture_mouse()
    set_depth_test(1)
    set_ambient(0.3)
    bind yaw = 0.0
    bind pitch = 0.0
    bind px = 0.0
    bind py = 0.0
    bind pz = -4.0
    bind t = 0.0
    while window_is_open() {
        bind yaw = yaw + mouse_dx() * 0.003
        bind pitch = pitch + mouse_dy() * 0.003
        if pitch > 1.5 { bind pitch = 1.5 }
        if pitch < -1.5 { bind pitch = -1.5 }
        bind fx = sin(yaw) * 0.08
        bind fz = cos(yaw) * 0.08
        if key_down("w") { bind px = px + fx  bind pz = pz + fz }
        if key_down("s") { bind px = px - fx  bind pz = pz - fz }
        if key_down("d") { bind px = px + fz  bind pz = pz - fx }
        if key_down("a") { bind px = px - fz  bind pz = pz + fx }
        set_camera(cos(yaw), sin(yaw), cos(pitch), sin(pitch))
        set_camera_pos(px, py, pz)
        fill(8, 10, 24)
        clear_lights()
        add_light(-3, -6, 0, 1.0, 0.95, 0.85, 1.0, 0)
        vtex_grid(0, 2, 8,  1,0,0,  0,0,1,  20, 20, 1, 1,  t, 0)
        set_color(240, 150, 110)
        cube(0, 0, 8, 1,1,1, t * 0.5, t, 0, 0)
        present()
        bind t = t + 0.02
    }
}
```

Lighting set-up

Lights belong to the camera chapter because the same per-frame discipline applies: clear, add, draw. How the lights shade a surface is explained in the next chapter.

`set_ambient(level)`

Sets the ambient light that fills everything regardless of the lights.

- **Aliases:** zh `环境光` ; ja `環境光設定` ; ko `환경광설정` ; th `ตั้งแสงรอบข้าง`
- **Parameters:** `level` from 0 to 1 (default 0.15).
- **Returns:** nothing.
- **Notes:** The default shading model also tints unlit areas with a cool shadow colour; see `set_shadow_color` .

```
GUI
set_ambient(0.25)
```

`add_light(x, y, z, r, g, b, intensity, radius)`

Adds a coloured point light at a world position for this frame.

- **Aliases:** zh `加灯` ; ja `ライト追加` ; ko `조명추가` ; th `เพิ่มแสง`
- **Parameters:** `x` , `y` , `z` position (defaults 0, -3, 3). `r` , `g` , `b` colour, either 0 to 1 or 0 to 255 (if any channel exceeds 1.5 all three are treated as 0 to 255). `intensity` brightness multiplier (default 1). `radius` distance at which the light fades to nothing; 0 means no falloff (default 0).
- **Returns:** nothing.
- **Notes:** The argument order is position, **colour**, intensity, radius. Some older pages in the repository show intensity before colour; the source is as listed here. Lights accumulate until you call `clear_lights()` , so clear and re-add them each frame.

```
GUI
clear_lights()
add_light(0, -5, 4,  1.0, 0.85, 0.6,  1.2, 20)
```

`clear_lights()`

Removes every light.

- **Aliases:** zh `清灯` ; ja `ライト消去` ; ko `조명초기화` ; th `ล้างแสง`
- **Parameters:** none.
- **Returns:** nothing.

```
GUI
clear_lights()
```

`set_fog(r, g, b, start, end)`

Distance fog: geometry fades toward a colour between two camera depths.

- **Aliases:** ko `안개설정` ; th `ตั้งหมอก` ; zh/ja `雾` , `霧設定`
- **Parameters:** `r` , `g` , `b` fog colour (0 to 255, clamped); `start` depth where fog begins; `end` depth where it is complete. `end <= 0` turns fog off.
- **Returns:** nothing.

```
GUI
set_fog(8, 10, 24, 10, 40)
```

CHAPTER 21

Shapes, Vector Textures, Pixel Textures and Effects

This chapter lists the builtins that put things into the 3D scene: the parametric solids (`cube`, `sphere`, `gear`, and about two dozen more), the low-level triangle and line calls, the `vtex_*` vector patterns, the `tex_*` pixel textures, and the shading, post-processing and light-volume effects that give the output its "anime holographic" look.

Everything here assumes the window, frame loop and camera of the previous chapter. In the short fragments, the names `n` and `frame` stand for a frame counter that the surrounding loop maintains. All snippets in this chapter need an open window; they are marked `ling-gui` and were checked against the interpreter's dispatch table. The pictures were produced by running the programs shown (or close variants) and calling `screenshot()`.

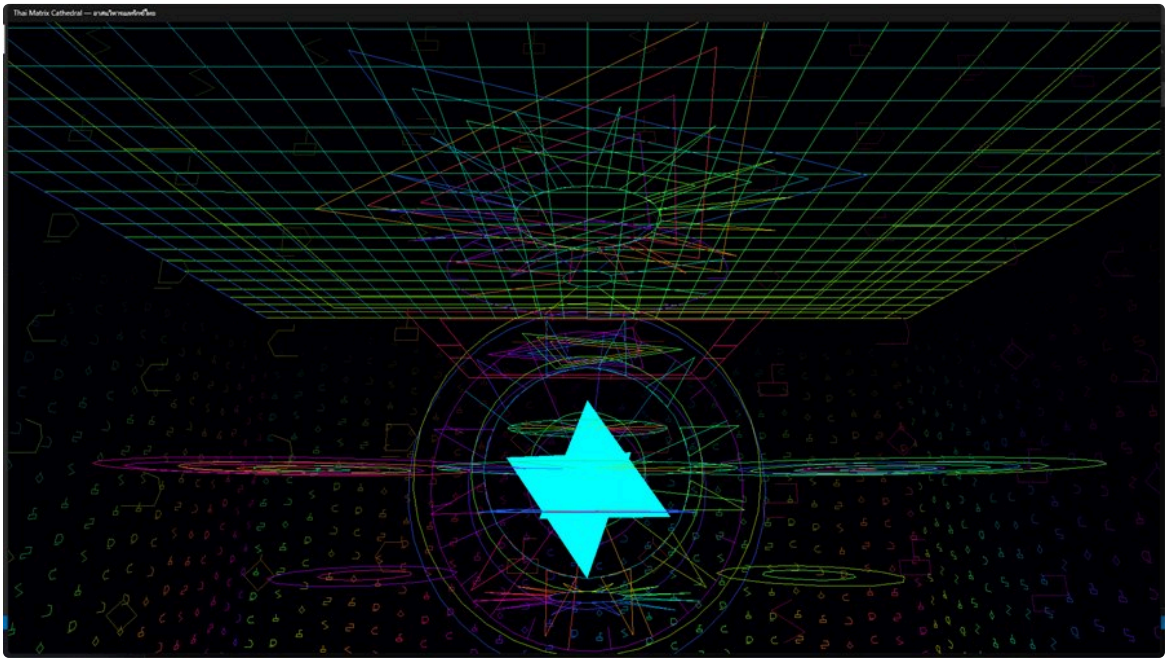


Figure 11.1. A room assembled from `vtex_*` patterns and a few solid shapes, drawn live. Source: `examples/3d-rooms/3d-room.ling` in the Ling repository.

Part 1. Primitive solids

Every solid shares one call shape:

```
name(cx, cy, cz,  sx, sy, sz,  rx, ry, rz,  mode,  e0, e1, e2)
```

Group	Meaning
`cx, cy, cz`	World position of the centre
`sx, sy, sz`	Scale per axis (default 1 each). Each shape is built about one unit in radius (the cube spans -1 to 1), so the scale is roughly the **half-extent** or radius
`rx, ry, rz`	Euler rotation in radians (default 0), applied in the order X, then Y, then Z, after scaling
`mode`	`0` filled (default), `1` wireframe, `2` both
`e0, e1, e2`	Shape-specific extras. `0` or omitted selects the default; a value that is not positive also selects the default

Only the position needs to be given; every other argument has a default. Trailing arguments may be dropped.

The pen colour (`set_color`) is the base colour for the fill and the colour of the wireframe. Solids are lit by `add_light` and `set_ambient`, shaded by the model described in Part 4, depth-sorted with the rest of the 3D queue, and drawn when `present()` is called.

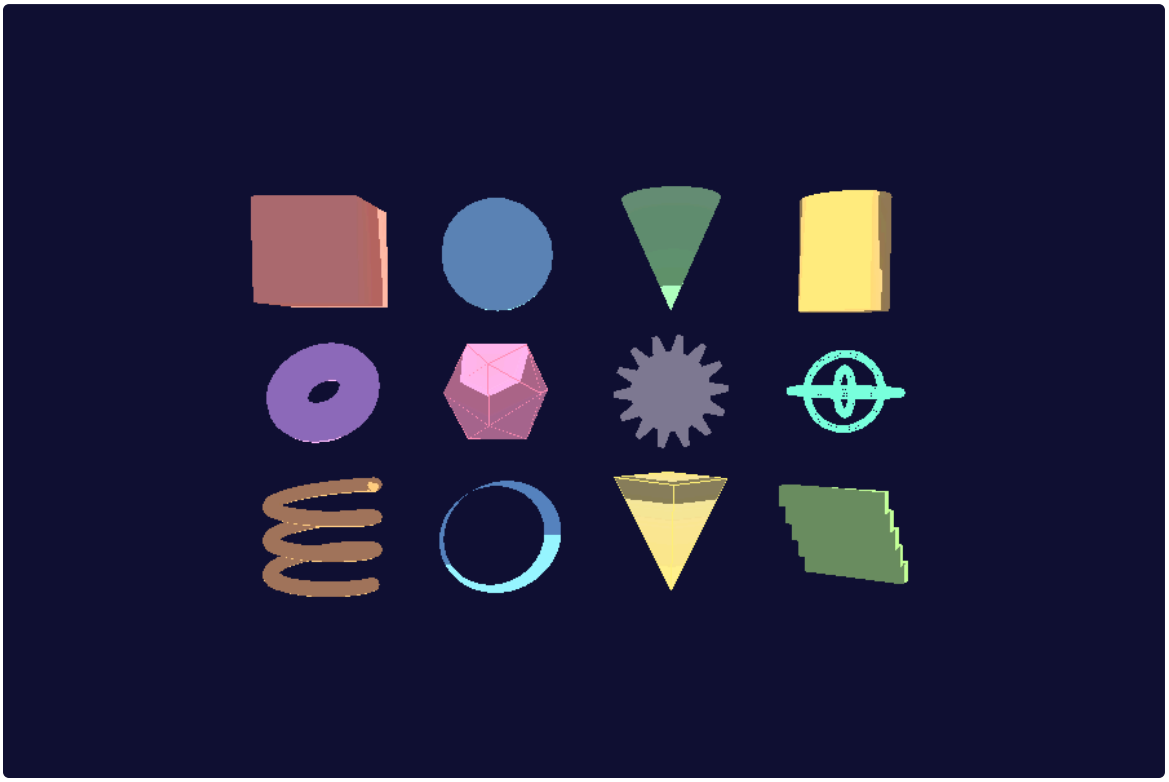


Figure 11.2. Twelve of the solids, lit by one warm light. Programs that want the pen colours to show through faithfully set `set_shade_mode(1)`, `set_rim(0, 0, 0, 0)` *and a dark* `set_shadow_color` *first; see Part 4.*

Remember the y axis points down (chapter 10). A `cone` has its apex at +**y**, so with the default camera it points down the screen; rotate by pi on x to flip it. A `gear` lies in the x-z plane, so rotate it by about 1.57 on x to face the camera.

```
GUI
bind start = do {
    open_window(900, 600, "Shapes")
    set_depth_test(1)
    set_ambient(0.25)
    set_shade_mode(1)
    set_rim(0, 0, 0, 0)
    set_shadow_color(30, 30, 50)
    bind n = 0
    while window_is_open() {
        fill(8, 10, 24)
        set_camera(1, 0, 1, 0)
        clear_lights()
        add_light(-2, -6, -2,  1.0, 0.95, 0.85,  0.6, 0)
        bind a = n * 0.03
        set_color(230, 120, 90)
        cube(-2.8, 0, 7.5,  0.8,0.8,0.8,  a, a * 0.7, 0,  0)
        set_color(120, 200, 255)
        sphere(0, 0, 7.5,  0.9,0.9,0.9,  0,0,0,  0)
        set_color(190, 190, 200)
        gear(2.8, 0, 7.5,  0.9,0.3,0.9,  1.57, a * 2, 0,  0,  14, 0.3)
        present()
        bind n = n + 1
    }
}
```

Summary: solids

Group	Builtins
Round	`cube`, `sphere`, `icosphere`, `dome`, `cylinder`, `cone`, `capsule`, `torus`
Prisms and pyramids	`pyramid`, `prism`, `frustum`
Dice	`tetrahedron`, `octahedron`, `dodecahedron`, `icosahedron`
Mechanical	`gear`, `gyro`, `arch`, `stairs`
Curves and surfaces	`helix`, `spring`, `star_prism`, `capsule_chain`, `mobius`

`cube(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode)`

An axis-aligned box. Non-uniform scales give any rectangular block.

- **Aliases:** en `box`; zh `方块`; ja `箱`; ko `상자`, `정육면체`; th `กล่อง`, `ลูกบาศก์`; zh/ja `立方体`
- **Parameters:** the common signature. No extras.
- **Returns:** nothing.
- **Notes:** The edge length is twice the scale: `cube(0,0,8, 1,1,1, ...)` is a 2-unit cube.

```
GUI
set_color(230, 120, 90)
cube(0, 0, 8,  1, 1, 1,  0, 0.6, 0,  0)
```

`sphere(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments, rings)`

A UV sphere; unequal scales make an ellipsoid.

- **Aliases:** zh `球体`; ja `球`; ko `구`; th `ทรงกลม`
- **Parameters:** the common signature; `segments` around the equator (default 16, range 3 to 128); `rings` pole to pole (default 12, range 2 to 128).
- **Returns:** nothing.

```
GUI
set_color(120, 200, 255)
sphere(0, 0, 8,  1, 1, 1,  0, 0, 0,  0,  32, 24)
```

`icosphere(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, subdivisions)`

A geodesic sphere made by subdividing an icosahedron; its triangles are more even than those of a UV sphere.

- **Aliases:** zh `二十面球`; ja `アイコ球`; ko `아이코구체`; th `ทรงกลมเหลี่ยม`
- **Parameters:** the common signature; `subdivisions` 0 to 4 (default 1).
- **Returns:** nothing.

```
GUI
icosphere(0, 0, 8,  1, 1, 1,  0, 0, 0,  2,  2)
```

`dome(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments, rings)`

A hemisphere with a closed base, for cupolas and skies.

- **Aliases:** zh `穹顶`; ja `ドーム`; ko `돔`; th `โดม`
- **Parameters:** the common signature; `segments` (default 24); `rings` (default 8).
- **Returns:** nothing.

```
GUI
dome(0, 1, 8,  2, 2, 2,  0, 0, 0,  0)
```

`cylinder(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments)`

A capped tube. The height runs along local y from -1 to 1, so it is `2 * sy` tall.

- **Aliases:** zh `圆柱`; ja `円柱`; ko `원기둥`; th `ทรงกระบอก`
- **Parameters:** the common signature; `segments` (default 24, range 3 to 256).
- **Returns:** nothing.
- **Notes:** For a flat disc or plate use a small `sy`, for example 0.05.

```
GUI
cylinder(0, 0, 8,  0.8, 1.2, 0.8,  0, 0, 0,  0)
```

`cone(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments)`

A capped cone with its apex at +y and base at -y.

- **Aliases:** zh `圆锥`; ja `円錐`; ko `원뿔`; th `กรวย`
- **Parameters:** the common signature; `segments` (default 24).
- **Returns:** nothing.
- **Notes:** Because +y is down the screen, an unrotated cone points downward. Use `rx = 3.14159` to point it up.

```
GUI
cone(0, 0, 8,  0.8, 1, 0.8,  3.14159, 0, 0,  0)    # apex up
```

`capsule(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments, rings)`

A cylinder with hemispherical caps: a pill.

- **Aliases:** zh `胶囊` ; ja `カプセル` ; ko `캡슐` ; th `แคปซูล`
- **Parameters:** the common signature; `segments` (default 16); `rings` per cap (default 6).
- **Returns:** nothing.

```
GUI
capsule(0, 0, 8,  0.5, 1, 0.5,  0, 0, 0.5,  0)
```

`torus(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments, sides, tube)`

A ring (donut) lying in the x-z plane.

- **Aliases:** en `ring` ; zh `圆环` ; ja `トーラス` ; ko `토러스` ; th `ทอรัส`
- **Parameters:** the common signature; `segments` around the ring (default 32); `sides` around the tube (default 12); `tube` tube radius as a fraction of the outer radius (default 0.35, range 0.02 to 0.9).
- **Returns:** nothing.
- **Notes:** `ring` is an English synonym.

```
GUI
torus(0, 0, 8,  1.2, 1.2, 1.2,  1.0, 0, 0,  0,  48, 16, 0.25)
```

`pyramid(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, sides)`

An N-sided pyramid; the default four sides is the classic square pyramid.

- **Aliases:** zh `金字塔` ; ja `ピラミッド` ; ko `피라미드` ; th `พีระมิด`
- **Parameters:** the common signature; `sides` (default 4).
- **Returns:** nothing.

```
GUI
pyramid(0, 0, 8,  1, 1, 1,  3.14159, 0.5, 0,  2)
```

`prism(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, sides)`

An extruded N-sided polygon: three sides for a triangular prism, six for a hexagonal column.

- **Aliases:** zh `棱柱` ; ja `角柱` ; ko `각기둥` ; th `ปริซึม`
- **Parameters:** the common signature; `sides` (default 6).
- **Returns:** nothing.

```
GUI
prism(0, 0, 8,  1, 1, 1,  0, 0.4, 0,  0,  3)
```

`tetrahedron(...)` , `octahedron(...)` , `dodecahedron(...)` , `icosahedron(...)`

The four Platonic solids besides the cube: dice with 4, 8, 12 and 20 faces. They take the common signature and no extras.

- **Aliases,** `tetrahedron` **:** en `d4` ; ko `정사면체` ; th `ทรงสี่หน้า` ; zh/ja `四面体`
- **Aliases,** `octahedron` **:** en `d8` ; ko `정팔면체` ; th `ทรงแปดหน้า` ; zh/ja `八面体`
- **Aliases,** `dodecahedron` **:** en `d12` ; ko `정십이면체` ; th `ทรงสิบสองหน้า` ; zh/ja `十二面体`
- **Aliases,** `icosahedron` **:** en `d20` ; ko `정이십면체` ; th `ทรงยี่สิบหน้า` ; zh/ja `二十面体`
- **Parameters:** the common signature, no extras.
- **Returns:** nothing.
- **Notes:** `d4` , `d8` , `d12` and `d20` are English synonyms. Use `mode` 2 on a bright colour for crisp gems.

```
GUI
set_color(255, 140, 180)
icosahedron(0, 0, 8,  1, 1, 1,  0.4, 0.7, 0,  2)
```

`gear(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, teeth, tooth)`

An extruded spur gear lying in the x-z plane; the thickness comes from `sy` .

- **Aliases:** en `cog` ; zh `齿轮` ; ja `歯車` ; ko `톱니바퀴` ; th `เฟือง`
- **Parameters:** the common signature; `teeth` count (default 12, range 3 to 96); `tooth` radial tooth depth as a fraction of the radius (default 0.25, range 0.02 to 0.6).
- **Returns:** nothing.

- **Notes:** Rotate by about 1.57 on `rx` so the face looks at the camera. `cog` is an English synonym.

```
GUI
gear(0, 0, 8,  1, 0.3, 1,  1.57, 0, n * 0.05,  0,  16, 0.3)
```

`helix(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, turns, tube, sides)` and `spring(...)`

A tube swept along a helix around the y axis. `spring` is a tighter, thinner coil with different defaults.

- **Aliases,** `helix`**:** zh `螺旋线`; ja `らせん`; ko `나선`; th `เกลียว`
- **Aliases,** `spring`**:** zh `弹簧`; ja `ばね`; ko `스프링`; th `สปริง`
- **Parameters:** the common signature; `turns` (default 3 for `helix`, 6 for `spring`); `tube` radius (0.15 and 0.12); `sides` around the tube (default 8).
- **Returns:** nothing.

```
GUI
helix(0, 0, 8,  0.8, 0.8, 0.8,  0, n * 0.03, 0,  0,  4, 0.12, 8)
```

`draw_triangle_3d(ax,ay,az, bx,by,bz, cx,cy,cz)`

Draws one lit triangle in the pen colour.

- **Aliases:** en `triangle3d`; th `วาดสามเหลี่ยม3มิติ`
- **Parameters:** three world-space points.
- **Returns:** nothing.
- **Notes:** Lighting uses the face normal from the winding order, so the visible side depends on the order of the points. After `set_material` the triangle uses the material model (Part 4).

```
GUI
set_color(255, 220, 120)
draw_triangle_3d(-1, 1, 8,  1, 1, 8,  0, -1, 8)
```

`draw_quad_3d(x0,y0,z0, ... )` , `draw_pent_3d` , `draw_hex_3d` , `draw_polygon_3d`

Draw a convex polygon, fan-triangulated, lit and fogged like `draw_triangle_3d`.

- **Aliases:** en `quad3d` , `draw_pent_3d` , `pent3d` , `draw_hex_3d` , `hex3d` , `draw_polygon_3d` , `polygon3d` ; th `วาดรูปหลายเหลี่ยม3มิติ` , `วาดสี่เหลี่ยม3มิติ` , `วาดหกเหลี่ยม3มิติ` , `วาดห้าเหลี่ยม3มิติ`
- **Parameters:** the vertices as consecutive `x, y, z` triples (`draw_quad_3d` takes 12 numbers, `draw_pent_3d` 15, `draw_hex_3d` 18); or a single argument, a flat list `[x0,y0,z0, x1,y1,z1, ...]` . At most 8 vertices are used. The four names share one implementation: the vertex count comes from how many triples you pass.
- **Returns:** nothing.

```
GUI
set_color(120, 200, 255)
draw_quad_3d(-2, 1, 8,  2, 1, 8,  2, 1, 12,  -2, 1, 12)      # a floor tile
draw_polygon_3d([0,-1,8,  1,0,8,  0,1,8,  -1,0,8])           # a diamond
```

`draw_line_3d(ax,ay,az, bx,by,bz)`

A depth-sorted line between two world points, clipped at the near plane.

- **Aliases:** en `line3d` ; zh `画3D线` ; ja `3D線描く` ; ko `3D선그리기` ; th `วาดเส้น3มิติ`
- **Parameters:** two world-space points.
- **Returns:** nothing.
- **Notes:** Shared edges between solids are de-duplicated automatically. See `set_antialias` for smooth lines.

```
GUI
set_color(255, 255, 255)
draw_line_3d(-2, 0, 8,  2, 0, 8)
```

Summary: vtex

Builtin	Pattern
`vtex_grid`	Rectangular grid
`vtex_rings`	Concentric polygons with twist
`vtex_star`	Spinning star polygon

Builtin	Pattern
`vtex_spiral`	Archimedean spiral
`vtex_flower`	Flower of Life
`vtex_halftone`	Cross-hatch dither fill
`vtex_tessellated`	Wavy triangle mesh
`vtex_hyperbolic_uv`	Poincaré-disc ceiling grid
`vtex_letter_rain`	Thai-letter matrix rain
`vtex_lotus`	Lotus petals
`vtex_chakra`	Dhamma wheel
`vtex_yantra`	Interlocked triangles in a square
`vtex_spiked_cog`	Spiked gear
`vtex_torii`	Torii gate silhouette
`vtex_pagoda`	Tiered pagoda silhouette

In the entries below, `PLANE` stands for the nine plane arguments `cx,cy,cz, ux,uy,uz, vx,vy,vz`.

`vtex_grid(PLANE, cols, rows, cell_w, cell_h, fr, hue)`

A rectangular grid of lines, centred on the plane origin.

- **Aliases:** zh `纹格` ; ja `格子模様` ; ko `격자무늬` ; th `ลายตาราง`
- **Parameters:** `cols` , `rows` cell counts (default 10, 10); `cell_w` , `cell_h` cell size in world units (default 1, 1); `fr` , `hue` (default 0).
- **Returns:** nothing.
- **Notes:** The grid spans `cols * cell_w` by `rows * cell_h` . Colour varies along the lines.

```
GUI
vtex_grid(0, 3, 9,  1,0,0,  0,0,1,  12, 8, 1, 1,  frame, 0)
```

`vtex_rings(PLANE, n_rings, n_sides, max_r, twist, fr, hue)`

Concentric N-sided polygons; each ring is rotated by `twist` more than the previous one, giving a vortex.

- **Aliases:** zh `纹环` ; ja `同心円` ; ko `동심원` ; th `ลายวงซ้อน`

- **Parameters:** `n_rings` (default 6); `n_sides` per ring (default 6); `max_r` outermost radius (default 3); `twist` radians per ring (default 0); `fr`, `hue`.
- **Returns:** nothing.

```
GUI
vtex_rings(0, -1, 8,  1,0,0,  0,-1,0,  6, 6, 1.2, 0.2,  frame, 0)
```

`vtex_star(PLANE, n_pts, r_out, r_in, rot_speed, fr, hue)`

A star polygon alternating between an outer and an inner radius. It spins as `fr` grows.

- **Aliases:** zh 纹星; ja 星模様; ko 별무늬; th ลายดาว
- **Parameters:** `n_pts` points (default 6); `r_out` (default 2); `r_in` (default 1); `rot_speed` radians of rotation per frame (default 0.01); `fr`, `hue`.
- **Returns:** nothing.

```
GUI
vtex_star(0, -1, 8,  1,0,0,  0,-1,0,  8, 1.2, 0.5, 0.02,  frame, 1)
```

`vtex_spiral(PLANE, n_turns, max_r, steps, fr, hue)`

An Archimedean spiral from the centre outward.

- **Aliases:** zh 纹螺; ja 螺旋; ko 나선; th ลายเกลียว
- **Parameters:** `n_turns` revolutions (default 3); `max_r` outer radius (default 3); `steps` line segments (default 120); `fr`, `hue`.
- **Returns:** nothing.

```
GUI
vtex_spiral(0, 1, 8,  1,0,0,  0,-1,0,  3, 1.2, 120,  frame, 2)
```

`vtex_flower(PLANE, radius, n_sides, fr, hue)`

The Flower of Life: seven overlapping circles, one in the centre and six around it, each drawn as an N-gon. The whole flower rotates slowly.

- **Aliases:** zh 纹花; ja 花模様; ko 꽃무늬; th ลายดอก

- **Parameters:** `radius` of each circle and the spacing between centres (default 1); `n_sides` per circle (default 24); `fr`, `hue`.
- **Returns:** nothing.

```
GUI
vtex_flower(0, -1, 8,  1,0,0,  0,-1,0,  1.2, 24,  frame, 2)
```

How a solid is lit

For each vertex the shader mixes a base colour (your pen colour) with:

1. an **ambient** term, `set_ambient`, blended with a **shadow tint** (`set_shadow_color`) wherever the surface faces away from every light;
2. each light's contribution: pen colour times a soft step ramp of the angle to the light, times the light colour, intensity and distance falloff;
3. a **rim glow** (`set_rim`), brighter where the surface turns away from the viewer; and
4. in the default `holo` mode, a faint iridescent sheen that depends on the surface normal.

The colour is interpolated across each triangle, then quantised into a few brightness **bands** per pixel. The result is smooth shading with crisp toon edges.

The defaults are deliberately stylised: a cool indigo shadow tint, a cyan rim at strength 0.6 and the holo sheen. They make unstyled scenes look cyan and washed out. For colours that match your pen faithfully, as in Figure 11.2, set `set_shade_mode(1)`, `set_rim(0, 0, 0, 0)` and a dark neutral `set_shadow_color`.

`set_shade_mode(m)`

Chooses the overall lighting style for solids.

- **Aliases:** zh `设置着色`; ja `シェード設定`; ko `셰이드모드`; th `ตั้งการแรเงา`
- **Parameters:** `m` `0` flat per-face cel, one band per triangle and visibly faceted; `1` smooth cel, toon bands without sheen and with the rim strength reduced to 40 percent; `2` holographic (default).
- **Returns:** nothing.

```
GUI
set_shade_mode(1)
```

`set_cel_bands(n)`

Sets the number of brightness bands.

- **Aliases:** zh `设置色阶` ; ja `セル段数` ; ko `셀밴드` ; th `ตั้งระดับสี`
- **Parameters:** `n` at least 2 (default 4). Fewer bands look chunkier.
- **Returns:** nothing.

```
GUI
set_cel_bands(3)
```

`set_shadow_color(r, g, b)`

Sets the tint used in unlit regions, so shadows can be coloured rather than black.

- **Aliases:** zh `设置阴影色` ; ja `影の色` ; ko `그림자색` ; th `ตั้งสีเงา`
- **Parameters:** `r` , `g` , `b` from 0 to 255 (default 26, 33, 77).
- **Returns:** nothing.

```
GUI
set_shadow_color(28, 36, 90)       # cool indigo
```

`set_rim(strength, r, g, b)`

Sets the Fresnel rim glow around silhouettes.

- **Aliases:** zh `设置边缘光` ; ja `リム設定` ; ko `림라이트` ; th `ตั้งขอบเรือง`
- **Parameters:** `strength` (default 0.6; 0 turns the rim off); `r` , `g` , `b` rim colour from 0 to 255 (default 115, 217, 255).
- **Returns:** nothing.

```
GUI
set_rim(0.7, 120, 220, 255)
```

`toon_outlines(thickness, color, threshold)`

Draws ink outlines where depth changes sharply.

- **Aliases:** zh `卡通轮廓` ; ja `トゥーンアウトライン` ; ko `툰아웃라인` ; th `ตั้งเส้นขอบการ์ตูน`
- **Parameters:** `thickness` half-width in pixels (0 is off, 1.5 is the anime default); `color` a packed `0xRRGGBB` number (0 is black, the default); `threshold` the depth jump that counts as an edge (default 0.05).
- **Returns:** nothing.

```
GUI
set_depth_test(1)
toon_outlines(1.5, 0, 0.05)
```

`light_pool(x, y, z, radius, r, g, b, intensity)`

A soft additive radial splash of light on the floor at height `y` : the coloured pool a lamp throws on the ground.

- **Aliases:** zh `光池` ; ja `ライトプール` ; ko `빛웅덩이` ; th `แอ่งแสง`
- **Parameters:** `x` , `y` , `z` centre of the pool (default 0); `radius` (default 20); `r` , `g` , `b` colour from 0 to 255 (default 255); `intensity` , about 0.2 to 1.5 (default 1).
- **Returns:** nothing.
- **Notes:** The edge fades smoothly and is fog-aware. Use `set_blend(1)` while drawing volumetric light so it adds to the scene.

`light_beam(x, y, z, floor_y, radius, r, g, b, intensity)`

A volumetric god-ray: a soft additive double cone from the light position down to a floor plane, widening to `radius` .

- **Aliases:** zh `光柱` ; ja `ライトビーム` ; ko `빛기둥` ; th `ลำแสงไฟ`
- **Parameters:** `x` , `y` , `z` position of the light; `floor_y` height of the floor plane; `radius` width at the floor (default 14); `r` , `g` , `b` colour from 0 to 255; `intensity` (default 1).
- **Returns:** nothing.
- **Notes:** Pair it with `light_pool` at the base.

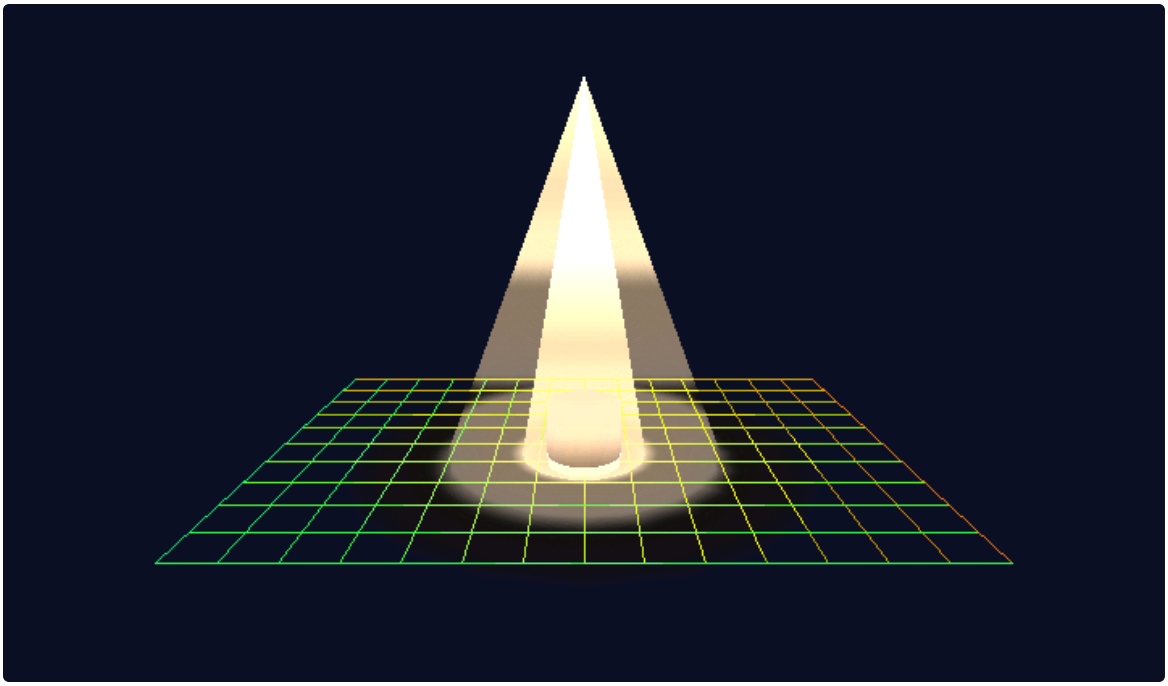


Figure 11.10. `light_beam` *and* `light_pool` *drawn with additive blending over a* `vtex_grid` *floor and a lit cylinder.*

```
GUI
bind start = do {
    open_window(720, 420, "Volumetric")
    set_depth_test(1)
    set_ambient(0.2)
    set_shade_mode(1)
    set_rim(0, 0, 0, 0)
    set_shadow_color(30, 30, 50)
    bind n = 0
    while window_is_open() {
        fill(6, 8, 18)
        set_camera(1, 0, cos(0.25), sin(0.25))
        set_camera_pos(0, -3, 0)
        clear_lights()
        add_light(0, -5, 8,  1.0, 0.9, 0.7,  0.8, 0)
        vtex_grid(0, 2, 10,  1,0,0,  0,0,1,  14, 10, 1, 1,  n, 0)
        set_color(220, 180, 120)
        cylinder(0, 1.3, 9,  0.8, 0.7, 0.8,  0, 0, 0,  0)
        set_blend(1)
        light_beam(0, -6, 9, 2, 3.0,  255, 220, 160,  0.8)
        light_pool(0, 2, 9, 5.0,  255, 220, 160,  0.9)
        set_blend(0)
        flush_3d()
        present()
        bind n = n + 1
    }
}
```

Summary: tex

Builtin	Pattern
`tex_checkerboard`	Two-colour checker
`tex_gradient`	Linear two-colour gradient
`tex_noise`	Fractal noise through a palette
`tex_spiral`	Animated spiral bands
`tex_ripple`	Concentric ripples

Builtin	Pattern
`tex_mandelbrot`	Mandelbrot set
`tex_julia`	Julia set
`tex_voronoi`	Voronoi cells
`tex_halftone`	Halftone dots
`tex_freq_map`	Audio-reactive spectrum

`tex_checkerboard(x, y, w, h, tiles, r1,g1,b1, r2,g2,b2)`

A two-colour checker pattern.

- **Aliases:** th `ลายตารางหมากรุก`
- **Parameters:** `tiles` number of tiles across the width and the height (default 8); first colour `r1, g1, b1` (default 255, 255, 255); second colour `r2, g2, b2` (default 0, 0, 0).
- **Returns:** nothing.

```
GUI
tex_checkerboard(10, 10, 200, 150, 8,  240, 240, 240,  30, 30, 60)
```

`tex_gradient(x, y, w, h, angle_deg, r1,g1,b1, r2,g2,b2)`

A linear gradient from the first colour to the second along a direction.

- **Aliases:** th `ลายไล่สี`
- **Parameters:** `angle_deg` direction in degrees (default 0, left to right); first colour (default 0, 0, 0); second colour (default 255, 255, 255).
- **Returns:** nothing.

```
GUI
tex_gradient(220, 10, 200, 150, 45,  255, 80, 0,  0, 80, 255)
```

`tex_noise(x, y, w, h, scale, octaves, seed, palette)`

Fractal (fBm) noise mapped through a palette.

- **Aliases:** th `ลายนอยส์`

- **Parameters:** `scale` noise frequency across the rectangle (default 4); `octaves` layers of detail, 1 to 8 (default 4); `seed` (default 0); `palette` (default `"rainbow"`).
- **Returns:** nothing.

```
GUI
tex_noise(430, 10, 200, 150, 5, 4, 7, "fire")
```

CHAPTER 22

Vector Fonts, the UI Toolkit and Dialogs

This chapter covers everything that puts text and interface on the screen: real TrueType and OpenType fonts rendered as resolution-independent vectors, the holographic stroke font, the `ui_*` toolkit of meters, gauges and controls, and the cinematic dialog box.

All of these draw into the same framebuffer as the 2D calls of chapter 10, so they belong between `fill()` and `present()`. To draw them over a 3D scene, call `flush_3d()` first.

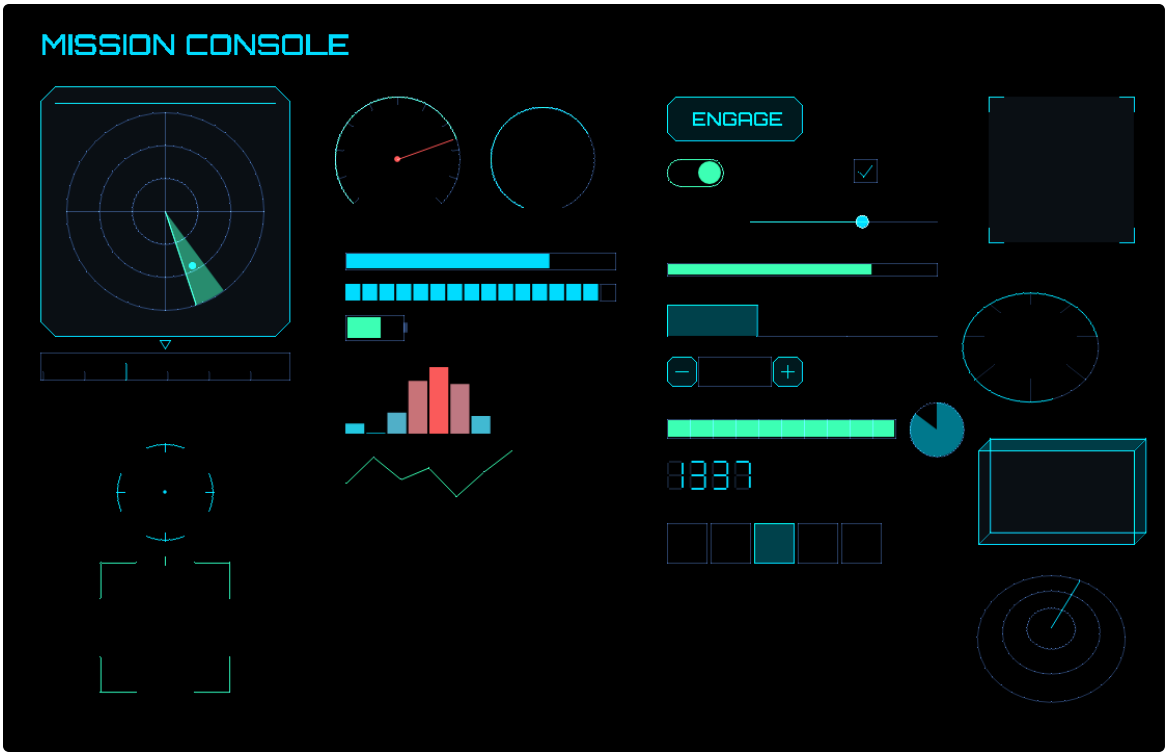


Figure 12.1. Every widget on this screen is a `ui_*` call, redrawn each frame in a few lines of Ling. Source: `examples/ui/hud_showcase.ling` in the Ling repository.

Part 1. Fonts and text

Ling draws two kinds of text.

- **Vector fonts.** `font_load` reads a TTF or OTF file; `font_text_fill`, `font_text` and `font_text_3d` draw its glyph outlines. The outlines keep their curves, so text is sharp at any size, in 2D or on a 3D plane. Latin, Chinese, Japanese, Korean and Thai all work if the font has the glyphs.
- **The built-in stroke font.** `ui_text` draws a monospaced single-stroke "holographic" face that needs no font file. It supports basic characters only.

The first time a glyph is used, its outline is extracted and cached on disk under `cache/fonts/<font>/<codepoint>.ling` (or `cache/fonts/<font>@<weight>/...` when a weight is given), relative to the directory the program runs in. Later runs read the cache.

Font loading is available in the native interpreter. In the web runtime `font_load` returns `-1`, so portable programs should check the handle and fall back to `ui_text`.

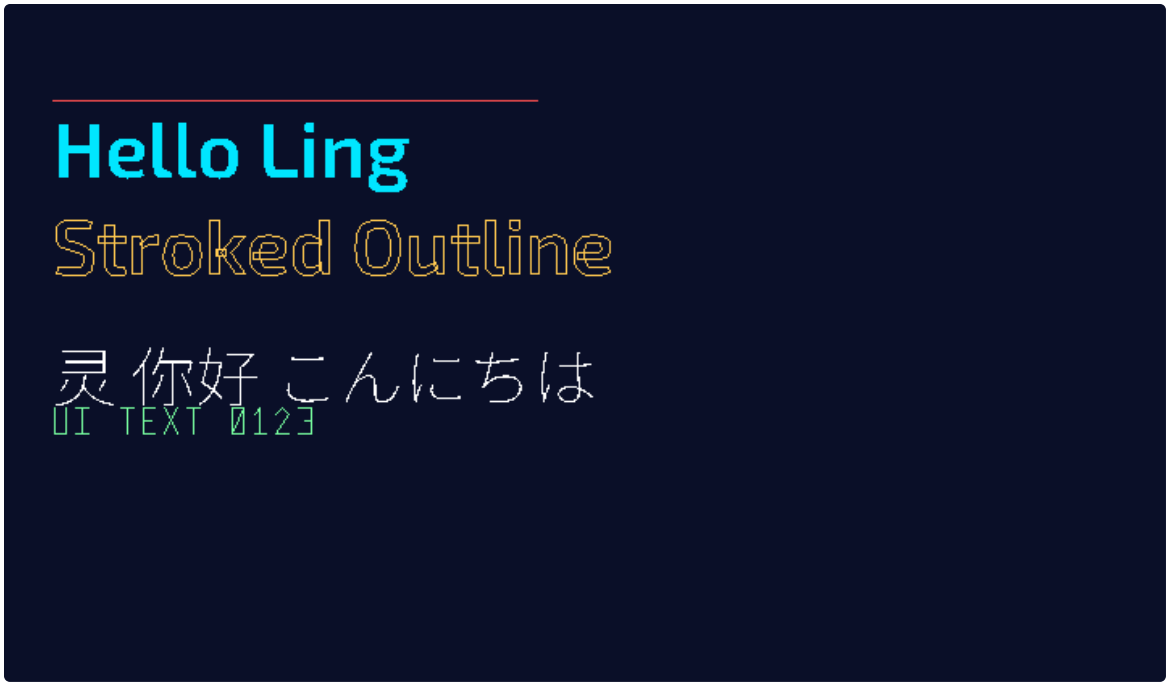


Figure 12.2. `font_text_fill` *(cyan),* `font_text` *(orange), a Chinese and Japanese line set in Noto Sans SC, and* `ui_text` *(green, bottom). The red line marks y = 60, the* `y` *given to the first line:* `y` *is the top of the text, so the glyphs hang below it.baseline.*

```
GUI
bind start = do {
    open_window(720, 420, "Fonts")
    bind f = font_load("assets/fonts/Exo2.ttf", 600)
    bind cjk = font_load("assets/fonts/NotoSansSC.ttf")
    while window_is_open() {
        fill(8, 10, 24)
        set_color(0, 230, 255)
        font_text_fill(f, 30, 60, 48, "Hello Ling")
        set_color(255, 200, 80)
        font_text(f, 30, 120, 48, "Stroked Outline")
        set_color(255, 255, 255)
        font_text_fill(cjk, 30, 200, 40, "灵 你好 こんにちは")
        set_color(120, 255, 160)
        ui_text(30, 250, 16, "UI TEXT 0123")
        present()
    }
}
```

Summary: fonts

Builtin	Purpose
`font_load(path, weight)`	Load a font, get a handle
`font_text_fill(font, x, y, px, text)`	Filled glyphs
`font_text(font, x, y, px, text)`	Stroked outlines
`font_text_3d(font, ..., size, text, rows)`	Text on a 3D plane
`font_width(font, px, text)`	Measure a string
`font_advance(font, char)`	Advance of one glyph
`font_glyph_outline(font, char, tol)`	Raw outline as polylines
`font_fallback(primary, fallback, scale)`	Per-character fallback font
`set_font_antialias(on)`, `get_font_antialias()`	Glyph smoothing
`ui_text(x, y, scale, text)`	Built-in stroke font

`font_load(path, weight)`

Loads a TrueType or OpenType font and returns a handle for the other `font_*` calls.

- **Aliases:** zh `加载字体` ; ja `フォント読込` ; ko `글꼴로드` ; th `โหลดฟอนต์`
- **Parameters:** `path` a file path. It is tried as given and then relative to the directory of the running script. `weight` optional variable-font weight, a positive number such as 600 for a solid, bold look; 0 or omitted uses the font's default.
- **Returns:** a handle (0 for the first font loaded, 1 for the next, and so on), or `-1` if the font could not be read. On failure a message such as `font_load failed (path): ...` is printed to the error stream.
- **Notes:** Check for `-1` and try a second path if your program may be started from different folders. The repository ships Exo2, Orbitron, Rajdhani, Noto Sans (SC, JP, KR, Thai) and Galmuri fonts in `assets/fonts/` .

```
GUI
bind f = font_load("assets/fonts/Exo2.ttf", 600)
if f < 0 { bind f = font_load("../../assets/fonts/Exo2.ttf", 600) }
```

`font_text_fill(font, x, y, px, text)`

Draws text as solid filled glyphs in the pen colour.

- **Aliases:** zh `填充字体` ; ja `フォント塗り` ; ko `글꼴채움` ; th `เติมฟอนต์`
- **Parameters:** `font` a handle from `font_load` ; `x` , `y` the top-left of the text line in pixels. Measured on screen, capital letters start about 0.3 `px` below `y` and the baseline lies about 0.97 `px` below `y` ; `px` the font size in pixels (default 16); `text` a string.
- **Returns:** nothing.
- **Notes:** Each glyph is filled independently so counters (the hole in "o") stay open. Edges are crisp unless you call `set_font_antialias(1)` . An invalid handle draws nothing.

```
GUI
set_color(0, 230, 255)
font_text_fill(f, 30, 60, 48, "Hello Ling")
```

`font_text(font, x, y, px, text)`

Draws text as stroked outlines in the pen colour and blend mode.

- **Aliases:** zh `字体文本`; ja `フォント文字`; ko `글꼴텍스트`; th `ข้อความฟอนต์`
- **Parameters:** as for `font_text_fill`.
- **Returns:** nothing.
- **Notes:** Cheaper than filled text and good for a wireframe look. Honours `set_blend` and `set_antialias`.

```
GUI
set_color(255, 200, 80)
font_text(f, 30, 120, 48, "Stroked Outline")
```

`font_text_3d(font, cx,cy,cz, ux,uy,uz, vx,vy,vz, size, text, fill_rows)`

Draws stroked text on a plane in 3D space.

- **Aliases:** zh `字体3D`; ja `フォント3D`; ko `글꼴3D`; th `ข้อความฟอนต์3มิติ`
- **Parameters:** `font` handle; `cx, cy, cz` the start of the text line; `ux, uy, uz` the direction the text advances (default 1, 0, 0); `vx, vy, vz` the up direction of the glyphs (default 0, 1, 0); `size` the height of one em in world units (default 1); `text` the string; `fill_rows` optional, when above 0 each glyph interior is filled with that many scan-line spans, producing true filled letters; 0 or omitted gives outlines only.
- **Returns:** nothing.
- **Notes:** The text flows through the depth-sorted line pipeline, so it rotates with the camera. Since +y is down the screen, upright text needs `V = (0, -1, 0)`, as in chapter 11.

```
GUI
set_color(120, 230, 255)
font_text_3d(f,  -2, 0, 8,   1,0,0,   0,-1,0,   0.6,  "LING 3D")
```

`font_width(font, px, text)`

Measures a string. To centre a label in a box `bx, by, bw, bh`, draw it at `x = bx + (bw - font_width(f, px, text)) / 2` and `y = by + bh / 2 - 0.64 * px`; the factor 0.64 puts the middle of the capital letters on the box's centre line.

- **Aliases:** zh `字体宽度`; ja `フォント幅`; ko `글꼴너비`; th `ความกว้างฟอนต์`
- **Parameters:** `font` handle; `px` size in pixels (default 16); `text` string. Note the order: size comes **before** the text, unlike the drawing calls.

- **Returns:** the width in pixels (0 for an invalid handle).
- **Notes:** Use it to centre or right-align text.

```
bind start = do {
    bind f = font_load("assets/fonts/Exo2.ttf", 600)
    println("width: " + str(font_width(f, 32, "Hello Ling")))
}
```

```
OUTPUT
width: 147.71200561523438
```

`ui_text(x, y, scale, text)`

Draws text in the built-in holographic stroke font. No font file is needed.

- **Aliases:** zh `界面文字`; ja `UI文字`; ko `UI텍스트`; th `ข้อความหน้าจอ`
- **Parameters:** `x`, `y` the top-left of the text; `scale` the glyph height in pixels, measured: glyphs fill rows `y` to `y + scale` (default 16); `text` the string.
- **Returns:** nothing.
- **Notes:** Drawn in the pen colour. Each glyph is `scale * 0.62` wide with a gap of `scale * 0.24`, so a string is about `len * scale * 0.86` pixels wide. Letters are drawn as capitals, and characters the face does not have (including all non-ASCII text) appear as a small box. It is not a substitute for a real font.

```
GUI
set_color(120, 255, 160)
ui_text(30, 250, 16, "UI TEXT 0123")
```

Part 2. The UI toolkit

The `ui_*` builtins are an **immediate-mode** toolkit. Nothing is stored between frames: you call the widget every frame, passing in the current value, and it draws itself and, for interactive controls, returns the new value. The program owns all state in ordinary variables.

```
GUI
bind start = do {
    open_window(480, 200, "Controls")
    bind volume = 0.4
    bind enabled = 0
    while window_is_open() {
        fill(6, 10, 22)
        bind volume = ui_slider(40, 60, 280, volume, 0, 1)       # in: value, out: new
value
        bind enabled = ui_toggle(40, 100, 52, 24, enabled)
        if ui_button(40, 140, 120, 36) == 1 { println("clicked") }
        present()
    }
}
```

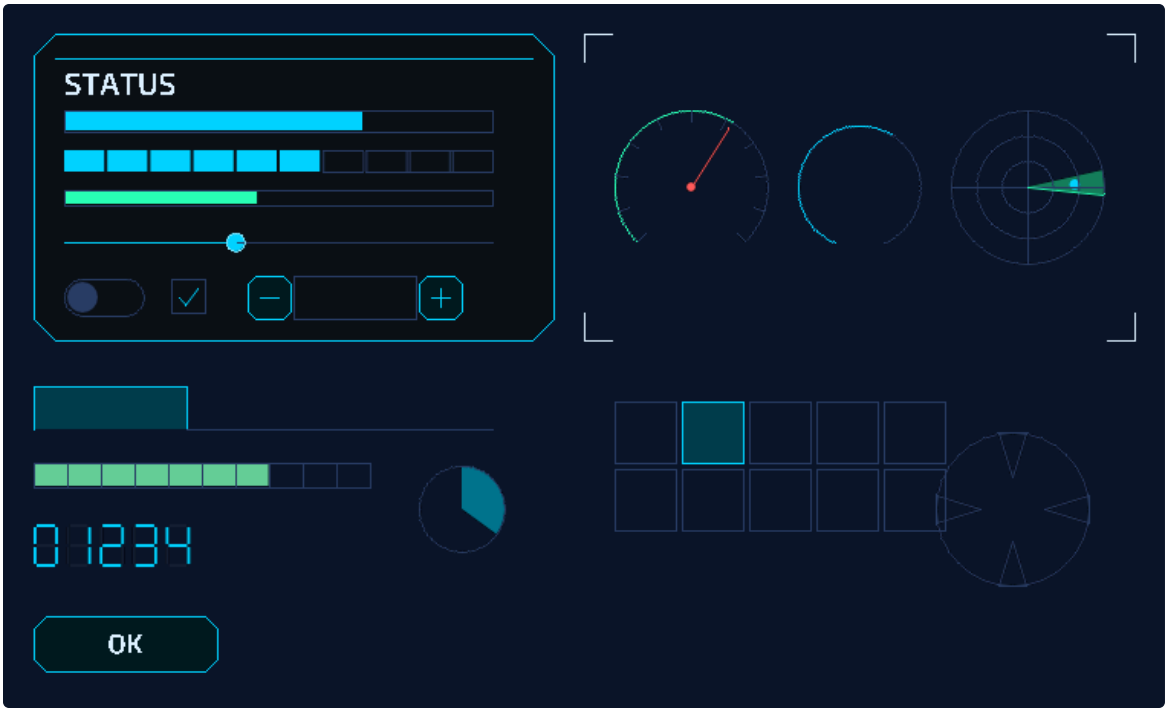


Figure 12.3. The widgets of this chapter on one screen: panel and frame, bar, segmented bar, progress, slider, toggle, checkbox, stepper, gauge, ring, radar, tabs, health bar, cooldown, counter, slot grid, d-pad and button.

Conventions shared by every widget:

- **Coordinates** are screen pixels, origin at the top-left, y down. Rectangular widgets take `x, y, w, h`; circular ones take a centre `cx, cy` and a radius `r`.
- **Colours** come from the current **theme** (below). Most widgets accept an optional trailing `r, g, b` triple (0 to 255) that overrides their main colour for one call; the position of that triple is given in each entry. Pass all three or none.
- **Widgets draw no text.** A button has no label, a tab strip has no captions and a stepper shows no number. Draw labels yourself with `ui_text` or `font_text_fill`. This keeps the toolkit independent of fonts.
- **Interaction** reads the left mouse button and the cursor. A control returns its new value; a button returns 1 on the single frame the mouse goes down over it. Because that edge is tracked across frames, `present()` must be called once per frame.
- **Blend.** Widgets honour the current blend mode, so `set_blend(1)` makes them glow.

Summary: UI toolkit

Group	Builtins
Theme and layout helpers	`ui_theme`, `ui_theme_colors`, `ui_hot`, `ui_frame`, `ui_bevel`, `ui_panel`, `ui_scanlines`
HUD	`ui_radar`, `ui_compass`, `ui_reticle`, `ui_target`
Meters	`ui_bar`, `ui_segbar`, `ui_gauge`, `ui_ring`, `ui_vu`, `ui_spark`, `ui_battery`, `ui_progress`
Controls	`ui_button`, `ui_toggle`, `ui_slider`, `ui_checkbox`, `ui_tabs`, `ui_stepper`, `ui_tooltip`
Game UI	`ui_healthbar`, `ui_cooldown`, `ui_counter`, `ui_minimap`, `ui_dpad`, `ui_slotgrid`, `ui_vignette`
Pseudo-3D	`ui_gauge3d`, `ui_panel3d`, `ui_radar3d`
Sound	`ui_sound`

`ui_theme(r,g,b, r,g,b, r,g,b, r,g,b, r,g,b, r,g,b)`

Sets the colours every `ui_*` widget draws from.

- **Aliases:** zh `界面主题`; ja `UIテーマ`; ko `인터페이스테마`; th `ธีมส่วนติดต่อ`
- **Parameters:** six colours as consecutive 0 to 255 triples, in this order: primary, accent, track (the dim background of meters), warn (alerts), text and bg (panel fill). Any triple you leave out keeps its current value, so `ui_theme(255, 0, 128)` changes only the primary.
- **Returns:** nothing.

- **Notes:** The built-in theme is primary `0, 210, 255`; accent `40, 255, 180`; track `44, 62, 100`; warn `255, 90, 90`; text `190, 235, 255`; bg `10, 16, 24`.

```
GUI
ui_theme(0,220,255,  60,255,180,  40,60,95,  255,90,90,  190,235,255,  10,16,24)
```

`ui_theme_colors()`

Reads the live theme back so that hand-drawn UI can match the toolkit.

- **Aliases:** ko `인터페이스테마색상`
- **Parameters:** none.
- **Returns:** a list of 18 numbers: `[pr,pg,pb, ar,ag,ab, tr,tg,tb, wr,wg,wb, xr,xg,xb, br,bg,bb]` for primary, accent, track, warn, text and bg.

```
bind start = do {
    ui_theme(255, 0, 128)
    println(ui_theme_colors())
}
```

```
OUTPUT
[255, 0, 128, 40, 255, 180, 44, 62, 100, 255, 90, 90, 190, 235, 255, 10, 16, 24]
```

`ui_hot(x, y, w, h)`

Is the mouse cursor inside a rectangle?

- **Aliases:** en `ui_hover`; zh `热区`; ja `ホットエリア`; ko `핫존`; th `พื้นที่สัมผัส`
- **Parameters:** the rectangle.
- **Returns:** boolean.
- **Notes:** Use it for hover effects and for hit-testing your own controls. It tests the cursor position reported by the window (0, 0 when there is no window).

```
GUI
if ui_hot(40, 140, 120, 36) { set_color(255, 255, 255) } else { set_color(120, 140,
160) }
ui_bevel(40, 140, 120, 36, 8)
```

`ui_frame(x, y, w, h, bracket)`

Sci-fi corner brackets around a rectangle, in the pen colour.

- **Aliases:** zh `边框` ; ja `フレーム枠` ; ko `프레임틀` ; th `กรอบ`
- **Parameters:** the rectangle; `bracket` the length of each bracket arm in pixels (default 14).
- **Returns:** nothing.

```
GUI
set_color(0, 210, 255)
ui_frame(380, 20, 360, 200, 18)
```

`ui_bevel(x, y, w, h, bevel)`

A bevelled (corner-cut) rectangle outline in the pen colour.

- **Aliases:** zh `斜角框` ; ja `ベベル枠` ; ko `베벨틀` ; th `กรอบเฉียง`
- **Parameters:** the rectangle; `bevel` the corner cut in pixels (default 10).
- **Returns:** nothing.

```
GUI
set_color(0, 210, 255)
ui_bevel(20, 20, 340, 200, 14)
```

`ui_panel(x, y, w, h, bevel, r, g, b)`

A bevelled panel with a faint fill and a header notch: the usual backdrop for a group of controls.

- **Aliases:** zh `面板` ; ja `パネル` ; ko `패널` ; th `แผง`
- **Parameters:** the rectangle (default size 200 by 120); `bevel` corner cut (default 12); optional border colour `r, g, b` at the sixth position (default the primary colour). The fill is the theme's `bg` .
- **Returns:** nothing.

```
GUI
ui_panel(20, 20, 340, 200, 14)
```

`ui_bar(x, y, w, h, value, max, r, g, b)`

A horizontal fill bar.

- **Aliases:** zh `进度条`; ja `バー`; ko `막대`; th `แถบ`
- **Parameters:** the rectangle (default 160 by 16); `value` the current value (default 0); `max` the value that fills the bar (default 1); optional fill colour at the seventh position.
- **Returns:** nothing.

```
GUI
ui_bar(40, 70, 280, 14, 70, 100)       # 70 percent
```

`ui_segbar(x, y, w, h, value, max, segments, r, g, b)`

A bar made of separate blocks.

- **Aliases:** zh `分段条`; ja `分割バー`; ko `분할막대`; th `แถบแบ่ง`
- **Parameters:** the rectangle; `value`, `max` as for `ui_bar`; `segments` the number of blocks (default 10); optional colour at the eighth position.
- **Returns:** nothing.

```
GUI
ui_segbar(40, 96, 280, 14, 6, 10, 10)    # six of ten blocks lit
```

`ui_gauge(cx, cy, r, value, max, r, g, b)`

A circular dial with a needle.

- **Aliases:** zh `仪表`; ja `ゲージ`; ko `게이지`; th `มาตรวัด`
- **Parameters:** centre; radius (default 50); `value`; `max` (default 1); optional needle colour at the sixth position (default the warn colour).
- **Returns:** nothing.

```
GUI
ui_gauge(450, 120, 50, 62, 100)
```

`ui_ring(cx, cy, r, value, max, r, g, b)`

A circular progress ring that fills clockwise.

- **Aliases:** zh `环表` ; ja `リングメーター` ; ko `링미터` ; th `วงแหวนวัด`
- **Parameters:** centre; radius (default 40); `value` ; `max` (default 1); optional colour at the sixth position.
- **Returns:** nothing.

```
GUI
ui_ring(560, 120, 40, 0.6, 1)
```

`ui_vu(x, y, w, h, levels, r, g, b)`

A bar-graph level meter drawn from a list of levels. Each bar blends from the fill colour toward the warn colour as its level rises.

- **Aliases:** zh `音量条` ; ja `VUメーター` ; ko `음량막대` ; th `มาตรเสียง`
- **Parameters:** the rectangle (default 160 by 60); `levels` a list of numbers; optional colour at the sixth position.
- **Returns:** nothing.
- **Notes:** Levels are clamped to the range 0 to 1. Combine with the `fft_bands` analyser for a spectrum display.

```
GUI
ui_vu(330, 330, 160, 60, [0.2, 0.5, 0.8, 1.0, 0.7, 0.3])
```

`ui_spark(x, y, w, h, values, r, g, b)`

A sparkline: a tiny line graph of a list of numbers, scaled automatically so the lowest value sits at the bottom and the highest at the top.

- **Aliases:** zh `迷你图` ; ja `スパークライン` ; ko `스파크라인` ; th `กราฟจิ๋ว`
- **Parameters:** the rectangle (default 160 by 40); `values` a list of at least two numbers (otherwise nothing is drawn); optional colour at the sixth position (default the accent colour).
- **Returns:** nothing.

```
GUI
ui_spark(330, 410, 160, 40, [3, 5, 4, 8, 6, 9, 7])
```

`ui_battery(x, y, w, h, value, max, r, g, b)`

A battery icon whose fill is proportional to the charge. It switches to the warn colour below 25 percent.

- **Aliases:** zh `电池` ; ja `バッテリー` ; ko `배터리` ; th `แบตเตอรี่`
- **Parameters:** the rectangle (default 50 by 22); `value` (default 1); `max` (default 1); optional colour at the seventh position.
- **Returns:** nothing.

```
GUI
ui_battery(40, 20, 50, 22, 0.8, 1)
```

`ui_progress(x, y, w, h, fraction, r, g, b)`

A thin progress bar for a fraction from 0 to 1.

- **Aliases:** zh `进度` ; ja `プログレス` ; ko `진행바` ; th `ความคืบหน้า`
- **Parameters:** the rectangle (default 200 by 12); `fraction` (default 0); optional colour at the sixth position (default the accent colour).
- **Returns:** nothing.

```
GUI
ui_progress(40, 122, 280, 10, 0.45)
```

`ui_button(x, y, w, h, r, g, b)`

A bevelled button that highlights on hover and darkens when pressed. It draws no label.

- **Aliases:** zh `按钮` ; ja `ボタン` ; ko `버튼` ; th `ปุ่ม`
- **Parameters:** the rectangle (default 120 by 40); optional colour at the fifth position.
- **Returns:** `1` on the one frame the left mouse button goes down over it, otherwise `0` .
- **Notes:** Draw a caption with `font_text_fill` or `ui_text` . Compare against 1: `if ui_button(...) == 1 { ... }` .

```
GUI
if ui_button(20, 400, 120, 36) == 1 { println("engage") }
set_color(220, 240, 255)
bind lw = font_width(ui, 18, "ENGAGE")
font_text_fill(ui, 20 + (120 - lw) / 2, 400 + 18 - 0.64 * 18, 18, "ENGAGE")   # centred
in the 120 by 36 button
```

`ui_toggle(x, y, w, h, state, r, g, b)`

A sliding on/off switch.

- **Aliases:** zh `开关` ; ja `トグル` ; ko `토글` ; th `สวิตช์`
- **Parameters:** the rectangle (default 52 by 24); `state` the current state, non-zero for on; optional on-colour at the sixth position.
- **Returns:** the new state, `1` or `0` . A click on it flips the state.

```
GUI
bind enabled = ui_toggle(40, 180, 52, 24, enabled)
```

`ui_slider(x, y, w, value, min, max, r, g, b)`

A horizontal slider.

- **Aliases:** zh `滑块` ; ja `スライダー` ; ko `슬라이더` ; th `แถบเลื่อน`
- **Parameters:** `x, y` the left end of the track; `w` its width (default 160); `value` the current value; `min` , `max` the range (default 0 and 1); optional colour at the seventh position. There is no height parameter; the grab area is 20 pixels tall.
- **Returns:** the new value. While the button is held over the slider, the value follows the cursor.

```
GUI
bind volume = ui_slider(40, 156, 280, volume, 0, 1)
```

`ui_checkbox(x, y, size, checked, r, g, b)`

A square check box.

- **Aliases:** zh `复选框` ; ja `チェックボックス` ; ko `체크박스` ; th `ช่องเลือก`
- **Parameters:** `x, y` the top-left; `size` the side length (default 20); `checked` non-zero when ticked; optional colour at the fifth position.
- **Returns:** the new state, `1` or `0` .

```
GUI
bind sound = ui_checkbox(110, 180, 22, sound)
```

`ui_tabs(x, y, w, h, count, active, r, g, b)`

A tab strip of `count` equal tabs with one highlighted. It draws no captions.

- **Aliases:** zh `标签页` ; ja `タブ` ; ko `탭` ; th `แท็บ`
- **Parameters:** the rectangle (default 240 by 28); `count` number of tabs (default 3); `active` the selected index from 0; optional colour at the seventh position.
- **Returns:** the active index, updated if a tab was clicked.
- **Notes:** Divide the width by `count` to place captions yourself.

```
GUI
bind tab = ui_tabs(20, 250, 300, 28, 3, tab)
```

`ui_stepper(x, y, w, h, value, step, r, g, b)`

A minus and plus control around a value box. It draws no number.

- **Aliases:** zh `步进器` ; ja `ステッパー` ; ko `스테퍼` ; th `ตัวปรับค่า`
- **Parameters:** the rectangle (default 120 by 28); `value` ; `step` the change per click (default 1); optional colour at the seventh position.
- **Returns:** the new value.

```
GUI
bind count = ui_stepper(160, 178, 140, 28, count, 1)
ui_text(210, 186, 14, str(count))
```

`ui_healthbar(x, y, w, h, value, max, pulse, r, g, b)`

A notched game health bar. The fill shifts from the warn colour at empty to the accent colour at full, and below 30 percent it can flash toward white.

- **Aliases:** zh `血条` ; ja `体力バー` ; ko `체력바` ; th `แถบพลังชีวิต`
- **Parameters:** the rectangle (default 180 by 16); `value` (default 1); `max` (default 1); `pulse` how far the low-health flash mixes toward white, 0 to 1 (default 0); optional colour at the eighth position.
- **Returns:** nothing.

```
GUI
ui_healthbar(20, 300, 220, 16, hp, 100, 0.5 + 0.5 * sin(time_now() * 8))
```

`ui_cooldown(cx, cy, r, fraction, r, g, b)`

An ability cooldown dial: a filled pie wedge covering `fraction` of the circle, measured clockwise from twelve o'clock.

- **Aliases:** zh `冷却` ; ja `クールダウン` ; ko `쿨다운` ; th `คูลดาวน์`
- **Parameters:** centre; radius (default 28); `fraction` (default 0); optional colour at the fifth position.
- **Returns:** nothing.

```
GUI
ui_cooldown(300, 330, 28, 0.35)
```

`ui_counter(x, y, digit_w, digit_h, value, digits, r, g, b)`

A seven-segment style numeric readout.

- **Aliases:** zh `计数器` ; ja `カウンター` ; ko `카운터` ; th `ตัวนับ`
- **Parameters:** `x, y` top-left; `digit_w` , `digit_h` size of one digit (default 14 by 24); `value` an integer; `digits` how many digits to show (default 4); optional colour at the seventh position.
- **Returns:** nothing.
- **Notes:** Unlit segments are shown dimly, as on a real display.

```
GUI
ui_counter(20, 340, 16, 28, 1234, 5)
```

`ui_sound(name)`

Plays a short synthesised interface sound.

- **Aliases:** zh `界面音` ; ja `UI音` ; ko `인터페이스음` ; th `เสียงปุ่ม`
- **Parameters:** `name` one of `"click"` (the default for any other name), `"hover"`, `"confirm"`, `"toggle"`, `"tick"`, `"error"`.
- **Returns:** nothing.
- **Notes:** Silent if the audio engine is not running. Pair it with a button: `if ui_button(...) == 1 { ui_sound("click") }`.

```
GUI
if ui_button(20, 400, 120, 36) == 1 { ui_sound("confirm") }
```

Part 3. Dialog boxes

A cinematic text box in the style of a console adventure: text types out letter by letter, pages advance on a key press, names, places and items are colour-coded, and the world is paused while it is open. Only one dialog exists at a time.



Figure 12.4. After `dialog_show` and a few `dialog_step` calls. "Link" is a name (gold), "Master Sword" an item (green) and "Hyrule" a place (blue). The triangle at the lower right shows the page is complete and waiting for `dialog_advance`.

Markup inside the text. `{n}...{/}` marks a name, `{p}...{/}` a place and `{i}...{/}` an item; `\n` is a line break and `||` a page break.

The dialog is driven by the program, one call per frame:

```
GUI
bind start = do {
    open_window(720, 300, "Dialog")
    bind ui = font_load("assets/fonts/Exo2.ttf", 600)
    bind line = "Hey {n}Link{/}! Take the {i}Master Sword{/} to {p}Hyrule{/}
field.||Beware the night."
    while window_is_open() {
        fill(10, 14, 24)
        if key_pressed("space") {
            if dialog_active() { dialog_advance() } else { dialog_show(line, 34) }
        }
        if dialog_active() {
            dialog_step(0.016)                    # typewriter; the world is "paused"
            dialog_draw(30, 120, 660, 150, ui)
        }
        present()
    }
}
```

The example `examples/ui/dialog_demo.ling` in the repository is a complete scene built on this pattern.

Summary: dialogs

Builtin	Purpose
`dialog_show(text, cps)`	Start a dialog
`dialog_step(dt)`	Advance the typewriter
`dialog_advance()`	Reveal the page, or go to the next page, or close
`dialog_active()`	Is a dialog open?
`dialog_typing()`	Is text still being revealed?

Builtin	Purpose
`dialog_close()`	Close it now
`dialog_color(role, r, g, b)`	Recolour a text role
`dialog_draw(x, y, w, h, font)`	Draw the box and text

`dialog_show(text, cps)`

Starts a dialog, replacing any dialog already open.

- **Aliases:** zh `对话显示` ; ja `会話表示` ; ko `대화표시` ; th `แสดงบทสนทนา`
- **Parameters:** `text` the string with markup; `cps` typing speed in characters per second (default 32).
- **Returns:** nothing.

```
GUI
dialog_show("Welcome, {n}traveller{/}.", 40)
```

`dialog_step(dt)`

Advances the typewriter and any auto-advance timers. Call it once per frame while the dialog is active.

- **Aliases:** zh `对话步进` ; ja `会話更新` ; ko `대화스텝` ; th `ก้าวบทสนทนา`
- **Parameters:** `dt` seconds since the previous frame (default 0.016).
- **Returns:** nothing.

```
GUI
dialog_step(0.016)
```

`dialog_advance()`

The "press A" action: if the page is still typing, reveal the rest of it; otherwise go to the next page; after the last page, close the dialog.

- **Aliases:** zh `对话推进` ; ja `会話送り` ; ko `대화진행` ; th `เลื่อนบทสนทนา`
- **Parameters:** none.
- **Returns:** nothing.

- **Notes:** Wire it to a key press edge (`key_pressed`) so a held key does not skip pages.

```
GUI
if key_pressed("space") { dialog_advance() }
```

`dialog_active()`

True while a dialog is open.

- **Aliases:** zh `对话激活`; ja `会話中`; ko `대화중`; th `บทสนทนาทำงาน`
- **Parameters:** none.
- **Returns:** boolean.
- **Notes:** Use it to gate your world update so physics and input are suspended while the player reads.

```
GUI
if dialog_active() == 0 { bind px = px + 1 }       # game update runs only when no
dialog is open
```

`dialog_typing()`

True while the current page is still being revealed.

- **Aliases:** zh `对话打字`; ja `会話タイプ中`; ko `대화타이핑`; th `กำลังพิมพ์บทสนทนา`
- **Parameters:** none.
- **Returns:** boolean.

```
GUI
if dialog_typing() { ui_sound("tick") }
```

`dialog_close()`

Closes the dialog immediately.

- **Aliases:** zh `对话关闭`; ja `会話閉じる`; ko `대화닫기`; th `ปิดบทสนทนา`
- **Parameters:** none.
- **Returns:** nothing.

```
GUI
dialog_close()
```

`dialog_draw(x, y, w, h, font)`

Draws the bevelled box and the revealed text.

- **Aliases:** zh `对话绘制` ; ja `会話描画` ; ko `대화그리기` ; th `วาดบทสนทนา`
- **Parameters:** `x` , `y` , `w` , `h` the box (default 40, 0, 720, 150); `font` an optional `font_load` handle; with `-1` or omitted the stroke font is used.
- **Returns:** nothing.
- **Notes:** Call it every frame while the dialog is active. The page-complete marker animates with the program clock.

```
GUI
dialog_draw(50, 420, 800, 150, ui)
```

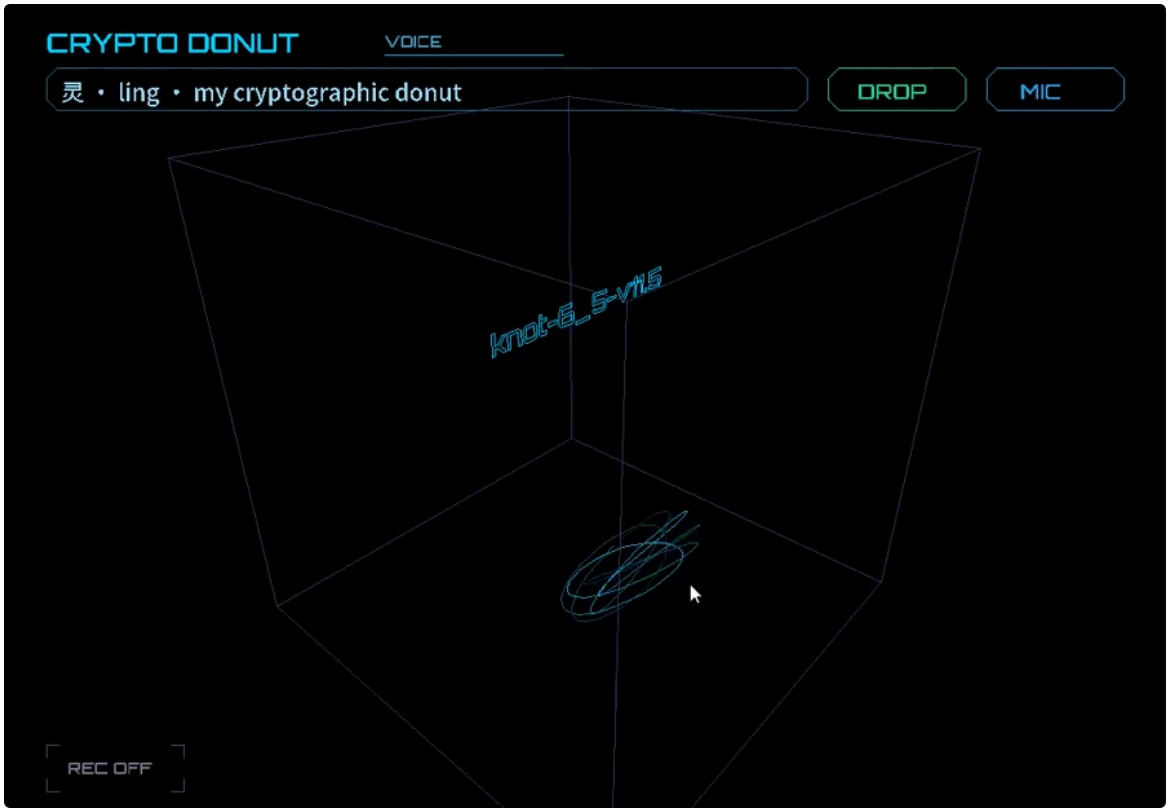


Figure 12.5. A complete application interface drawn with bevelled frames, vector fonts and buttons over a 3D wireframe scene. Source: `examples/crypto/crypto_donut_physics.ling`.

CHAPTER 23

Animation: Anima

Anima is Ling's animation system. It rests on a simple idea: any moving thing, whether a 2D drawing, a 3D mesh, a swarm of particles or a handful of numbers, is driven by small **drivers** that you call every frame with the current time. A driver is a function from time (and a few settings) to a number. You feed the number into a size, a position, an angle or a pattern argument.

Anima divides its drivers along one axis that it calls **temperament**:

- **Organic** drivers (the Chinese character 灵, *ling*, "spirit") produce lifelike, slightly soft motion: breathing, swaying, walking, springy follow-through, reaching with two-bone inverse kinematics.
- **Mechanical** drivers (机, *ji*, "machine") produce exact, rigid motion: meshed gears, cams, slider-cranks, rack and pinion.

A character can mix both: a creature with a breathing chest and a mechanical jaw.

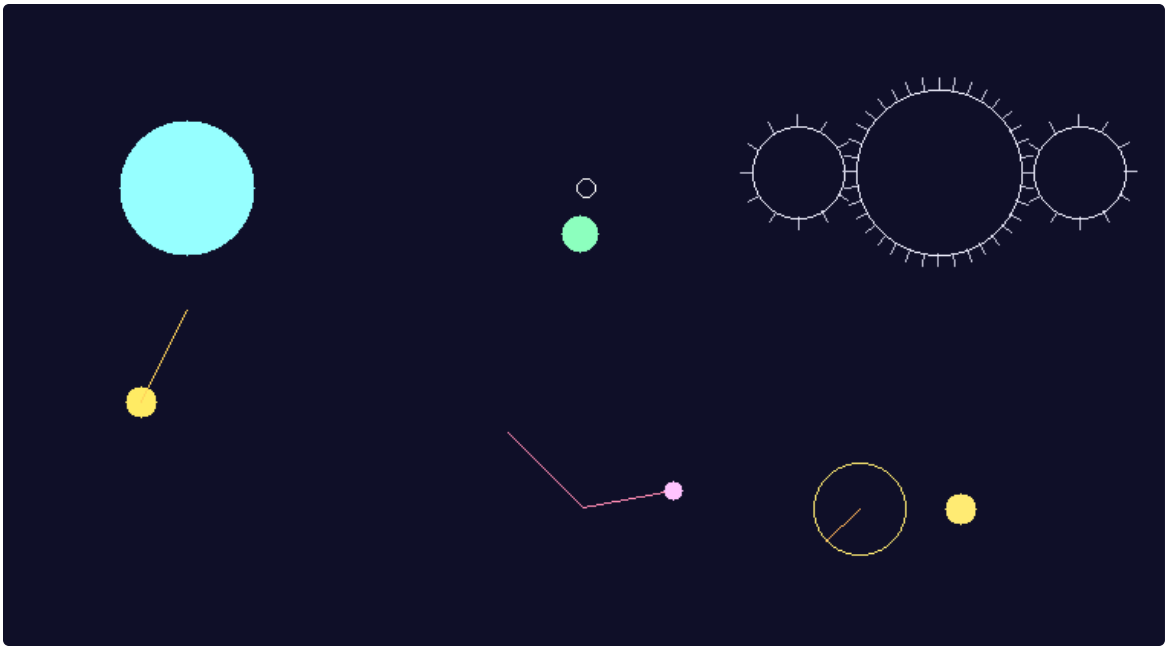


Figure 13.1. Every motion here is one Anima call per frame. Upper left: `breathe`. Below it: `gait_swing` and `gait_lift` moving a foot. Centre: `spring_to` chasing a moving target (the white ring). Lower centre: `ik2` bending a two-bone arm toward a target. Upper right: `gear_train` with 12, 36 and 12 teeth. Lower right: `piston` driven by a crank angle.

How to use a driver

Drivers are **pure functions**: they keep no hidden state, so calling one twice with the same arguments gives the same answer. Two consequences follow.

1. For time-based drivers (`breathe`, `wobble`, `gait_*`) you pass the current time yourself, usually the output of `time_now()` or a variable you add `1/60` to each frame. Nothing is "playing" behind your back.
2. For dynamic drivers (`spring_to`) the state is returned to you and you pass it back in next frame. The program owns the state.

Angles are in radians. Lengths and offsets are in whatever units your drawing uses.

```
fn r3(x: number) -> number {
    round(x * 1000) / 1000
}

bind start = do {
    println("tween(0, 10, 0.25)             = " + str(tween(0, 10, 0.25)))
    println("tween(0, 10, 2.0)              = " + str(tween(0, 10, 2.0)))
    println("tween_ease cubic_in at 0.5     = " + str(r3(tween_ease(0, 100, 0.5,
"cubic_in"))))
    println("tween_ease bounce_out at 0.5   = " + str(r3(tween_ease(0, 100, 0.5,
"bounce_out"))))
    println("breathe(0.25, 1, 0.2)          = " + str(r3(breathe(0.25, 1, 0.2))))
    println("wobble(0.25, 1, 2, 0)          = " + str(r3(wobble(0.25, 1, 2, 0))))
    println("gait_phase(0.3, 2)             = " + str(r3(gait_phase(0.3, 2))))
    println("gait_swing(0.125, 1, 30)       = " + str(r3(gait_swing(0.125, 1, 30))))
    println("gait_lift(0.125, 1, 10)        = " + str(r3(gait_lift(0.125, 1, 10))))
    bind sp = spring_to(0, 0, 1, 120, 14, 0.0166667)
    println("spring_to one step             = [" + str(r3(sp[0])) + ", " +
str(r3(sp[1])) + "]")
    bind a = ik2(1, 1, 1, 1)
    println("ik2(1, 1, 1, 1)                = [" + str(r3(a[0])) + ", " + str(r3(a[1]))
+ "]")
    println("gear_couple(3, 12, 36)         = " + str(r3(gear_couple(3, 12, 36))))
    println("gear_train(1, [12, 36, 12])    = " + str(gear_train(1, [12, 36, 12])))
    println("cam_lift(3.14159265, 2)         = " + str(r3(cam_lift(3.14159265, 2))))
    println("piston(0, 1, 3)                = " + str(r3(piston(0, 1, 3))))
    println("rack(2, 0.5)                   = " + str(rack(2, 0.5)))
    println("ease(quad_in, 0.5)             = " + str(ease("quad_in", 0.5)))
    println("ease(bounce, 0.5)              = " + str(r3(ease("bounce", 0.5))))
}
```

```
OUTPUT
tween(0, 10, 0.25)            = 2.5
tween(0, 10, 2.0)             = 10
tween_ease cubic_in at 0.5    = 12.5
tween_ease bounce_out at 0.5  = 76.563
breathe(0.25, 1, 0.2)         = 1.2
wobble(0.25, 1, 2, 0)         = 2
gait_phase(0.3, 2)            = 0.6
gait_swing(0.125, 1, 30)      = 21.213
gait_lift(0.125, 1, 10)       = 7.071
spring_to one step            = [0.033, 2]
ik2(1, 1, 1, 1)               = [1.571, -1.571]
gear_couple(3, 12, 36)        = -1
gear_train(1, [12, 36, 12])   = [1, -0.33333334326744 08, 1]
cam_lift(3.14159265, 2)        = 2
piston(0, 1, 3)               = 4
rack(2, 0.5)                  = 1
ease(quad_in, 0.5)            = 0.25
ease(bounce, 0.5)             = 0.766
```

The program above defines a small helper `r3` to round results to three decimals, because the drivers work in single precision and print long fractions. The output was produced by running it.

Summary: Anima builtins

Builtin	Temperament	Purpose
`tween(a, b, t)`	neutral	Linear interpolation
`tween_ease(a, b, t, curve)`	neutral	Eased interpolation
`ease(curve, t)`	neutral	Evaluate an easing curve
`breathe(t, rate, depth)`	organic	Breathing scale
`wobble(t, freq, amp, phase)`	organic	Sway or jiggle
`gait_phase(t, speed)`	organic	Walk-cycle phase
`gait_swing(t, speed, stride)`	organic	Leg swing
`gait_lift(t, speed, height)`	organic	Foot lift

Builtin	Temperament	Purpose
`spring_to(pos, vel, target, k, d, dt)`	organic	Spring step
`ik2(l1, l2, tx, ty)`	organic	Two-bone inverse kinematics
`gear_couple(angle, teeth_in, teeth_out)`	mechanical	Two meshed gears
`gear_train(angle, teeth)`	mechanical	A chain of meshed gears
`cam_lift(angle, lift)`	mechanical	Cam follower
`piston(angle, crank, rod)`	mechanical	Slider-crank
`rack(angle, radius)`	mechanical	Rack and pinion

Easing curves

Two builtins evaluate easing curves, and they use **different sets of names**. Both ignore case; unknown names fall back as noted.

`tween_ease` (Anima curves). Names may be written with or without underscores. Unknown names give `linear`.

Family	Names
Basic	`linear`, `step`
Quadratic	`quad_in`, `quad_out`, `quad_in_out`
Cubic	`cubic_in`, `cubic_out`, `cubic_in_out` (also `smooth`)
Sine	`sine_in`, `sine_out`, `sine_in_out`
Exponential	`expo_in`, `expo_out`, `expo_in_out`
Elastic	`elastic_in`, `elastic_out` (also `elastic`)
Back	`back_in`, `back_out`, `back_in_out` (also `overshoot`)
Bounce	`bounce_in`, `bounce_out` (also `bounce`)

`ease` (the UI toolkit's curves). Dashes and underscores are interchangeable. Unknown names give `quad_inout`.

Name	Meaning
`linear`	Straight line
`quad_in`, `ease_in`	Slow start
`quad_out`, `ease_out`	Slow finish
`quad_inout`, `ease`, `smooth`	Slow start and finish (the default)

Name	Meaning
`cubic_in`, `cubic_out`, `cubic_inout`	Stronger versions
`quart_out`	Very fast start, long slow finish
`sine`, `sine_inout`	Gentle sine in-out
`expo`, `expo_out`	Exponential decay
`back`, `back_out`	Overshoots, then settles
`elastic`, `elastic_out`	Springy overshoot
`bounce`, `bounce_out`	Bounces at the end

Note that `tween_ease` spells the in-out forms `quad_in_out` while `ease` spells them `quad_inout`; since `tween_ease` removes underscores before matching, both spellings work there.

Interpolation entries

`tween(a, b, t)`

Linear interpolation from `a` to `b`.

- **Aliases:** zh `补间`; ja `補間`; ko `트윈`; th `แทรกค่า`
- **Parameters:** `a` start value; `b` end value; `t` progress, **clamped** to the range 0 to 1.
- **Returns:** `a + (b - a) * t`.
- **Notes:** Because `t` is clamped, you can keep increasing it past 1 and the value stays at `b`.

```
GUI
bind x = tween(100, 700, (time_now() - t0) / 2.0)     # slides over two seconds
draw_disc(x, 300, 20)
```

`tween_ease(a, b, t, curve)`

Interpolation with an easing curve.

- **Aliases:** zh `缓动补间`; ja `緩和補間`; ko `이징트윈`; th `แทรกนุ่ม`
- **Parameters:** `a`, `b` start and end; `t` progress 0 to 1; `curve` an Anima curve name (see above), default `"linear"`.
- **Returns:** number. Elastic and back curves can overshoot `b` or undershoot `a`.

```
GUI
bind y = tween_ease(500, 200, p, "bounce_out")      # drops and bounces
draw_disc(400, y, 24)
```

`ease(curve, t)`

The raw value of a UI-toolkit easing curve at progress `t`: 0 at the start, 1 at the end.

- **Aliases:** none (English name only)
- **Parameters:** `curve` a name from the `ease` table above (default `"ease"`); `t` progress, clamped to 0 to 1.
- **Returns:** number, usually between 0 and 1.
- **Notes:** The curve name comes **first**. Use it when you want the curve value itself, for example to blend colours: `mix_color(..., ease("smooth", t))`.

```
GUI
mix_color(255, 80, 0,  0, 120, 255,  ease("smooth", fade))
```

Organic drivers

`breathe(t, rate, depth)`

A breathing multiplier centred on 1.0: `1 + depth * sin(2 pi * rate * t)`.

- **Aliases:** ko `호흡`; th `หายใจ`; zh/ja `呼吸`
- **Parameters:** `t` time in seconds; `rate` breaths per second (default 1); `depth` the amplitude (default 0.1).
- **Returns:** a number near 1. Multiply a size by it.
- **Notes:** It also drives pattern arguments: breathe the radius of a `vtex_flower`.

```
GUI
set_color(120, 220, 255)
draw_disc(120, 120, 40 * breathe(t, 0.4, 0.12))
```

`wobble(t, freq, amp, phase)`

A sine sway for secondary motion: `amp * sin(2 pi * (freq * t + phase))`.

- **Aliases:** zh `摆动`; ja `揺れ`; ko `흔들림`; th `โยก`
- **Parameters:** `t` time; `freq` cycles per second (default 1); `amp` amplitude (default 1); `phase` a phase offset in cycles, so `0.25` is a quarter-cycle shift (default 0).
- **Returns:** a number between `-amp` and `amp`.
- **Notes:** Give each limb a different `phase` and the whole body sways out of lockstep.

```
GUI
bind sway = wobble(t, 0.7, 12, 0.0)
```

`gait_phase(t, speed)`

Where a walk cycle is, as a fraction from 0 up to (but not including) 1.

- **Aliases:** zh `步相`; ja `歩相`; ko `걸음위상`; th `เฟสก้าว`
- **Parameters:** `t` time; `speed` cycles per second (default 1).
- **Returns:** number in `[0, 1)`.

```
GUI
bind phase = gait_phase(t, 2.0)        # two strides per second
```

`gait_swing(t, speed, stride)`

The forward and backward swing of a leg: `stride * sin(2 pi * phase)`.

- **Aliases:** zh `步摆`; ja `歩振り`; ko `걸음흔들`; th `ก้าวแกว่ง`
- **Parameters:** `t`, `speed` as for `gait_phase`; `stride` the swing amplitude (default 1).
- **Returns:** a number between `-stride` and `stride`.
- **Notes:** For the other leg use `gait_swing(t + 0.5 / speed, ...)`, half a cycle later.

```
GUI
bind foot_x = 120 + gait_swing(t, 1.0, 30)
```

`gait_lift(t, speed, height)`

The height a foot is raised: positive during the swing half of the cycle and zero while it is planted.

- **Aliases:** zh `抬脚`; ja `足上げ`; ko `발들기`; th `ยกเท้า`
- **Parameters:** `t`, `speed`; `height` the maximum lift (default 1).
- **Returns:** number from 0 to `height`.
- **Notes:** The screen y axis points down, so subtract the lift: `foot_y = ground_y - gait_lift(...)`.

```
GUI
bind foot_y = 260 - gait_lift(t, 1.0, 20)
```

`spring_to(pos, vel, target, stiffness, damping, dt)`

One step of a damped spring that pulls `pos` toward `target`. Use it for follow-through, camera smoothing and tails.

- **Aliases:** zh `弹向`; ja `バネ寄せ`; ko `스프링이동`; th `สปริงไป`
- **Parameters:** `pos` the current position; `vel` the current velocity; `target` where it is heading; `stiffness` the spring constant (default 120); `damping` the friction (default 14); `dt` the time step in seconds (default 1/60).
- **Returns:** a two-element list `[new_pos, new_vel]`.
- **Notes:** Keep both numbers and feed them back next frame. With the defaults (stiffness 120, damping 14) and `dt` of 0.016, a value moving to a target 1 unit away reaches 99 percent of the distance in about a quarter of a second, overshoots by roughly 5 percent and has settled within a second.

```
GUI
bind r = spring_to(px, pv, target_x, 120, 14, 0.016)
bind px = r[0]
bind pv = r[1]
```

`gear_couple(angle, teeth_in, teeth_out)`

The angle of a gear meshed with another. Meshed gears turn in opposite directions, scaled by the tooth ratio.

- **Aliases:** zh `齿轮联动`; ja `歯車連動`; ko `기어연동`; th `เฟืองทด`
- **Parameters:** `angle` the driving gear's angle in radians; `teeth_in` its tooth count; `teeth_out` the driven gear's tooth count (both default 1).
- **Returns:** `-angle * teeth_in / teeth_out`, or 0 if `teeth_out` is 0.
- **Notes:** A 12-tooth gear driving a 36-tooth gear turns it a third as fast, the other way.

```
GUI
bind small = t * 2.0
bind large = gear_couple(small, 12, 36)
```

`gear_train(angle, teeth)`

The angle of every gear in a chain of meshed gears.

- **Aliases:** zh `齿轮组`; ja `歯車列`; ko `기어열`; th `ชุดเฟือง`
- **Parameters:** `angle` the first gear's angle in radians; `teeth` a list of tooth counts, first gear first.
- **Returns:** a list of angles, the same length as `teeth`. The first element is `angle` itself; each following gear is meshed with the previous one.
- **Notes:** An empty or non-list `teeth` returns an empty list.

```
GUI
bind g = gear_train(t * 2.0, [12, 36, 12])
draw_arc(520, 110, 30, g[0], g[0] + 1.0)
```

CHAPTER 24

Audio, Music and the DAW

This chapter documents the sound builtins of the Ling runtime. They fall into three layers that are independent of each other:

1. **Audio engine** (`audio_*`, `ui_sound`, `morph_note`): a real-time positional synthesiser and sample player with a master effect chain. Backed by the `ling-audio` crate.
2. **Music toolkit** (`music_*`): decoding, analysis, playback of whole tracks, a General-MIDI-style polyphonic synthesiser driven by `.ling` patch files, MIDI files, rhythm-game judging and karaoke helpers. Backed by `ling-music`.
3. **DAW** (`daw_*`, `fs_*`, `dlg_*`, `st_*`, `px_*`): the scripting surface of the LingStationX digital audio workstation. Backed by `ling-daw`.

Conventions used in every entry:

- Arguments shown with a default are optional. A missing argument takes the default listed; a wrong type silently becomes the default (numbers) or an empty string.
- All times are in **seconds** unless an entry says *beats*.
- The runtime opens the audio device when the interpreter starts. If no output device exists, the `audio_*` builtins silently do nothing and a line beginning `audio init failed (no sound)` is printed to standard error. Analysis builtins (`music_load`, `music_bpm`, …) and all `daw_*` editing and rendering builtins work without a device.
- Aliases: most `audio_*` and `music_*` builtins can also be called by a translated name. The entries list the Chinese (zh), Japanese (ja), Korean (ko) and Thai (th) names. Most of them additionally have Persian, Arabic, Hebrew, Urdu, French, German and Russian names, which are not listed here; see `src/runtime/mod.rs`. The `daw_*`, `st_*` and `px_*` families have no aliases.
- Entries are tagged B (beginner essential), I (intermediate) or A (advanced or rarely needed).

Audio engine

The audio engine owns one output stream. Four kinds of sound share it:

- **Tones**: long-lived oscillators in numbered slots, positioned in 4D space (`audio_tone`).

- **One-shots**: UI blips (`audio_blip`, `ui_sound`), positional sound effects (`audio_sfx`) and morph notes (`morph_note`).
- **Samples**: decoded files played back at a position or flat (`audio_sample_*`).
- **Background music**: one looping WAV (`audio_bgm`).

The listener is the camera. `audio_listener` takes the cosine and sine of the camera yaw and pitch, the same four numbers you pass to `set_camera`, and `set_camera_pos` moves the listener position too. Pan is equal-power left/right from the sound's camera-space X; level falls off exponentially with distance. The final mix is soft-clipped with `tanh`.

Limits: tone slots grow on demand, with 32 simultaneous blips, 64 SFX voices, 32 morph voices and 64 sample voices. When a pool is full the oldest voice is dropped.

Master level and background music

Builtin	Level	Summary
`audio_volume(level)`	B	Set the master volume.
`audio_bgm(path, volume)`	B	Load a WAV file and loop it as background music.
`audio_bgm_volume(level)`	B	Change the background-music volume without reloading it.

`audio_volume(level)`

Set the master volume.

- **Aliases:** zh `音量` · ko `음량` · th `ระดับเสียง`
- **Parameters:**
- `level` (number, default 0.7) - linear gain applied to everything the audio engine mixes. 1.0 is unity.
- **Returns:** nothing.
- **Notes:** Does not affect the `music_*` engine, which has its own `music_volume`.

```
GUI
bind start = do {
  audio_volume(0.5)
}
```

`audio_bgm(path, volume)`

Load a WAV file and loop it as background music.

- **Aliases:** zh `背景乐` · ja `BGM` · ko `배경음악` · th `เพลงพื้นหลัง`
- **Parameters:**
- `path` (string) - path to a `.wav` file, used exactly as given (it is not resolved against the script folder).
- `volume` (number, default 0.5) - playback level.
- **Returns:** nothing. A missing or unreadable file prints `audio: bgm load failed` to standard error and the scene continues in silence.
- **Notes:**
- Only WAV is accepted here (read through the `hound` library and resampled to the device rate). Use `audio_sample_load` or `music_load` for MP3, OGG, FLAC and AAC.
- A second call replaces the current background track.

```
GUI
bind start = do {
  audio_bgm("theme.wav", 0.4)
}
```

`audio_bgm_volume(level)`

Change the background-music volume without reloading it.

- **Aliases:** zh `背景乐音量` · ja `BGM音量` · ko `배경음악음량` · th `ระดับเสียงพื้นหลัง`
- **Parameters:**
- `level` (number, default 0.5)
- **Returns:** nothing. Does nothing if no background track was loaded.

```
GUI
bind start = do {
  audio_bgm("theme.wav", 0.4)
  audio_bgm_volume(0.15)
}
```

Positional tones and the listener

Builtin	Level	Summary
`audio_tone(slot, x, y, z, w, freq, amp, lfo_rate, lfo_depth)`	I	Create or retune a continuous positional oscillator.
`audio_listener(cos_yaw, sin_yaw, cos_pitch, sin_pitch)`	I	Set the listener orientation used for panning and attenuation.

`audio_tone(slot, x, y, z, w, freq, amp, lfo_rate, lfo_depth)`

Create or retune a continuous positional oscillator.

- **Aliases:** zh `音调` · ja `音調` · ko `음조` · th `เสียงโทน`
- **Parameters:**
- `slot` (integer, default 0) - tone slot. Reusing a slot updates that tone in place; frequency and amplitude glide to their new values, so you can call this every frame.
- `x, y, z` (numbers, default 0) - world position of the source.
- `w` (number, default 1.0) - fourth-dimension value. It drives a sub-oscillator at `freq * w * 0.007` Hz that cross-modulates the carrier. 0 disables the effect.
- `freq` (number, default 220) - carrier frequency in Hz.
- `amp` (number, default 0.15) - linear amplitude. Pass 0 to silence the slot.
- `lfo_rate` (number, default 0.5) - vibrato speed in Hz.
- `lfo_depth` (number, default 0.02) - vibrato depth as a fraction of `freq` (0.03 is plus or minus 3 percent).
- **Returns:** nothing.
- **Notes:** There is no builtin to delete a slot; set `amp` to 0.

```
GUI
bind start = do {
  audio_volume(0.6)
  audio_tone(0, 2.0, 0.0, 6.0, 1.0, 196.0, 0.12, 0.5, 0.01)
}
```

`audio_listener(cos_yaw, sin_yaw, cos_pitch, sin_pitch)`

Set the listener orientation used for panning and attenuation.

- **Aliases:** zh `音频监听` · ja `音声リスナー` · ko `오디오리스너` · th `ผู้ฟัง`

- **Parameters:**
- `cos_yaw, sin_yaw, cos_pitch, sin_pitch` (numbers, defaults 1, 0, 1, 0) - the same four values passed to `set_camera`.
- **Returns:** nothing.
- **Notes:** The listener position is updated separately by `set_camera_pos`.

```
GUI
bind start = do {
  bind yaw = 0.5
  audio_listener(cos(yaw), sin(yaw), 1.0, 0.0)
}
```

One-shot sounds

Builtin	Level	Summary
`audio_blip(freq, duration, wave, amp)`	B	Play a short interface blip.
`ui_sound(name)`	B	Play one of the built-in interface sounds.
`audio_sfx(x, y, z, w, freq, amp, duration, wave)`	I	Play a one-shot synthesised effect at a world position.
`morph_note(x, y, z, w, freq, amp, duration, material, morph)`	A	Play a "yin-yang" morph note that blends a physical model with a digital one.
`audio_stop_sfx()`	I	Stop every one-shot voice immediately.

`audio_blip(freq, duration, wave, amp)`

Play a short interface blip.

- **Aliases:** zh `提示音` · ja `ビープ音` · ko `효과음` · th `เสียงบี๊บ`
- **Parameters:**
- `freq` (number, default 660) - frequency in Hz.
- `duration` (number, default 0.08) - length in seconds; minimum 0.01.
- `wave` (string, default "sine") - `"sine"`, `"square"` (or `"sq"`), `"saw"` / `"sawtooth"`, `"tri"` / `"triangle"`, `"noise"` (or `"wn"`, `"ns"`). Matching is case-insensitive and an unknown name falls back to sine.
- `amp` (number, default 0.25)
- **Returns:** nothing.

- **Notes:** Fast attack, exponential decay. Not positional.

```
GUI
bind start = do {
  audio_blip(880.0, 0.06, "square", 0.2)
}
```

`ui_sound(name)`

Play one of the built-in interface sounds.

- **Aliases:** zh `界面音` · ja `UI音` · ko `인터페이스음` · th `เสียงปุ่ม`
- **Parameters:**
- `name` (string, default "click") - `"click"`, `"hover"`, `"confirm"` (two rising square blips), `"error"` (two low saw blips), `"toggle"`, `"tick"`. Any other name plays `"click"`.
- **Returns:** nothing.

```
GUI
bind start = do {
  ui_sound("confirm")
}
```

`audio_sfx(x, y, z, w, freq, amp, duration, wave)`

Play a one-shot synthesised effect at a world position.

- **Aliases:** zh `音效` · ja `空間効果音` · ko `공간효과음` · th `เสียงเอฟเฟกต์`
- **Parameters:**
- `x, y, z` (numbers, default 0) - source position.
- `w` (number, default 1.0) - fourth dimension, as for `audio_tone`.
- `freq` (number, default 440) - Hz.
- `amp` (number, default 0.3)
- `duration` (number, default 0.15) - seconds, minimum 0.01.
- `wave` (string, default "sine") - same names as `audio_blip`.
- **Returns:** nothing.

```
GUI
bind start = do {
  audio_sfx(1.5, 0.0, 4.0, 1.0, 330.0, 0.3, 0.2, "triangle")
}
```

`audio_stop_sfx()`

Stop every one-shot voice immediately.

- **Aliases:** zh `停止音效` · ja `効果音停止` · ko `효과음정지` · th `หยุดเอฟเฟกต์ทั้งหมด`
- **Returns:** nothing.
- **Notes:** Clears the SFX pool, the morph voices and all sample voices, so an effect tail cannot leak across a scene change. Continuous tones, the background track and `music_*` playback are not touched.

```
GUI
bind start = do {
  audio_stop_sfx()
}
```

Samples

Builtin	Level	Summary
`audio_sample_load(path)`	I	Decode an audio file into a sample slot.
`audio_sample_play(id, x, y, z, w, volume, loop, pitch)`	I	Play a loaded sample at a world position.
`audio_sample_play_flat(id, volume, loop, pitch)`	I	Play a sample with no positioning.
`audio_sample_stop(voice)`	I	Stop a playing sample voice.
`audio_sample_duration(id)`	I	Length of a loaded sample in seconds.

`audio_sample_load(path)`

Decode an audio file into a sample slot.

- **Aliases:** zh `载入采样` · ja `サンプル読込` · ko `샘플로드` · th `โหลดตัวอย่างเสียง`

- **Parameters:**
- `path` (string) - WAV, FLAC, OGG, MP3 or AAC. If the path does not exist as given, it is tried relative to the script's folder.
- **Returns:** sample id (integer, counting from 0), or `-1` on failure (the reason goes to standard error). Also `-1` if there is no audio device.
- **Notes:** The file is mixed down to mono; positional playback pans it.

```ling
GUI
bind start = do {
  bind hit = audio_sample_load("hit.wav")
  print(audio_sample_duration(hit))
}
```

`audio_sample_play(id, x, y, z, w, volume, loop, pitch)`

Play a loaded sample at a world position.

- **Aliases:** zh `播放采样` · ja `サンプル再生` · ko `샘플재생` · th `เล่นตัวอย่างเสียง`
- **Parameters:**
- `id` (integer) - id from `audio_sample_load`.
- `x, y, z` (numbers, default 0) - position; the sound pans and attenuates relative to the listener.
- `w` (number, default 1.0) - fourth dimension (accepted for symmetry, no audible effect on samples).
- `volume` (number, default 1.0)
- `loop` (number, default 0) - a value above 0.5 loops.
- `pitch` (number, default 1.0) - playback-rate multiplier: 2.0 is an octave up, 0.5 an octave down. A non-positive value is treated as 1.0.
- **Returns:** voice id (integer). `0` if the id is invalid or there is no audio device. Pass the voice id to `audio_sample_stop`.
- **Notes:** On the web target the argument order is `(id, x, y, z, w, volume, loop)`, with `w` ignored; pitch is not available there.

```
GUI
bind start = do {
  bind hit = audio_sample_load("hit.wav")
  bind v = audio_sample_play(hit, 0.0, 0.0, 3.0, 1.0, 0.8, 0, 1.0)
}
```

`audio_sample_play_flat(id, volume, loop, pitch)`

Play a sample with no positioning.

- **Aliases:** zh `播放采样平` · ja `サンプル再生平` · ko `샘플재생평` · th `เล่นตัวอย่างเสียงเรียบ`
- **Parameters:**
- `id` (integer)
- `volume` (number, default 1.0)
- `loop` (number, default 0) - above 0.5 loops.
- `pitch` (number, default 1.0)
- **Returns:** voice id, or `0` on failure.
- **Notes:** No distance falloff and no pan, so the level does not depend on the listener. Use it for narration, voice-over and UI cues; use `audio_sample_play` for world sounds. This builtin has no translated aliases.

```
GUI
bind start = do {
  bind line = audio_sample_load("narration.ogg")
  audio_sample_play_flat(line, 1.0, 0, 1.0)
}
```

`audio_sample_stop(voice)`

Stop a playing sample voice.

- **Aliases:** zh `停止采样` · ja `サンプル停止` · ko `샘플정지` · th `หยุดตัวอย่างเสียง`
- **Parameters:**
- `voice` (integer) - the voice id returned by a play call.
- **Returns:** nothing.

```
GUI
bind start = do {
  bind hum = audio_sample_load("hum.wav")
  bind v = audio_sample_play_flat(hum, 0.5, 1, 1.0)
  audio_sample_stop(v)
}
```

`audio_sample_duration(id)`

Length of a loaded sample in seconds.

- **Aliases:** zh `采样时长` · ja `サンプル長さ` · ko `샘플길이` · th `ความยาวตัวอย่างเสียง`
- **Parameters:**
- `id` (integer) - id from `audio_sample_load`.
- **Returns:** seconds as a number; `0` if the id is invalid. This builtin has no translated aliases.
- **Notes:** Computed from the sample's own rate, independent of the device rate. Use it to pace cut-scenes to the length of the clip that was actually loaded.

```
GUI
bind start = do {
  bind line = audio_sample_load("narration.ogg")
  print(audio_sample_duration(line))
}
```

Master effects

The master chain is delay, then reverb, then a low-pass filter. All three are off or neutral until you set them.

Builtin	Level	Summary
`audio_fx_delay(time, feedback, mix)`	I	Configure the master echo.
`audio_fx_reverb(mix)`	I	Set the master reverb amount.
`audio_fx_lowpass(cutoff)`	I	Set the master low-pass cutoff.

`audio_fx_delay(time, feedback, mix)`

Configure the master echo.

- **Aliases:** zh `回声` · ja `ディレイ効果` · ko `딜레이` · th `เสียงสะท้อน`
- **Parameters:**
- `time` (number, default 0.3) - delay time in seconds, capped by the internal buffer.
- `feedback` (number, default 0.3) - clamped to 0..0.95.
- `mix` (number, default 0.3) - wet level, clamped to 0..1. Use 0 to switch the echo off.
- **Returns:** nothing.

```
GUI
bind start = do {
  audio_fx_delay(0.35, 0.4, 0.25)
}
```

`audio_fx_reverb(mix)`

Set the master reverb amount.

- **Aliases:** zh `混响` · ja `リバーブ` · ko `리버브` · th `เสียงก้อง`
- **Parameters:**
- `mix` (number, default 0.3) - wet level, clamped to 0..1.
- **Returns:** nothing.

```
GUI
bind start = do {
  audio_fx_reverb(0.4)
}
```

`audio_fx_lowpass(cutoff)`

Set the master low-pass cutoff.

- **Aliases:** zh `低通滤波` · ja `ローパス` · ko `저역통과` · th `กรองความถี่ต่ำ`
- **Parameters:**
- `cutoff` (number, default 1.0) - clamped to 0..1. 1.0 is open; lower values muffle the mix, which gives an underwater effect. The filter glides to the new target.

- **Returns:** nothing.

```
GUI
bind start = do {
  audio_fx_lowpass(0.15)
}
```

Music toolkit

The `music_*` builtins are backed by the `ling-music` crate. They cover four jobs:

- **Decode and analyse** an audio file: `music_load`, `music_duration`, `music_bpm`, `music_key`, `music_onsets`, `music_beat_grid`, `music_fft`, `music_fft_groups`. These need no audio device.
- **Play** a decoded track: `music_play`, `music_pause`, `music_stop`, `music_seek`, `music_pos`, `music_volume`. The music engine opens its own output stream the first time it is needed, separate from the `audio_*` engine, and plays one track at a time.
- **Synthesise** notes from instrument patches: `music_patch`, `music_note`, `music_note_on`, `music_note_off`.
- **Game and karaoke helpers**: MIDI files, `.lrc` lyrics, pitch conversion, sung-pitch scoring and rhythm judging.

Track, MIDI and lyric handles are small integers counting from 0 in load order. A failed load returns `-1`. Passing an unknown handle to an analysis builtin returns `0`, an empty string or an empty list, never an error.

Pitch arguments (`music_note`, `music_note_on`, `music_note_off`, `music_hz`) accept either a MIDI number (60 is middle C, 69 is A4) or a note-name string such as `"C4"`, `"A#3"` or `"Db5"`. The octave number follows the scientific convention in which C4 is MIDI 60.

Paths given to `music_load`, `music_patch`, `music_lrc` and `music_midi_load` are used as given when the file exists, otherwise they are tried relative to the folder of the running script.

The next example generates its own audio, so you can run it without any asset files: it renders a two-bar drum loop with the DAW builtins, then analyses the resulting WAV.

Analysing a rendered loop

```
bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind drums = daw_track_add("drums")
  bind clip = daw_clip_add_midi(drums, 0, 8)
  bind beat = 0
  while beat < 16 {
    daw_note_add(drums, clip, beat * 0.5, 0.25, 36, 0.9)
    bind beat = beat + 1
  }
  daw_bounce("loop.wav", 0, 8, 0.0, 16, "wav", 0.5)

  bind track = music_load("loop.wav")
  print("track id: " + str(track))
  print("seconds: " + str(round(music_duration(track))))
  print("bpm: " + str(round(music_bpm(track))))
  print("onsets: " + str(len(music_onsets(track))))
  print("beats in grid: " + str(len(music_beat_grid(track))))
}
```

```
OUTPUT
track id: 0
seconds: 4
bpm: 120
onsets: 15
beats in grid: 8
```

Loading and analysis

Builtin	Level	Summary
`music_load(path)`	B	Decode an audio file into memory and return a track handle.
`music_duration(track)`	B	Length of a decoded track in seconds.
`music_bpm(track)`	I	Estimate the tempo of a track in beats per minute.
`music_key(track)`	I	Estimate the musical key of a track.
`music_onsets(track)`	I	Times at which new notes or hits begin.

Builtin	Level	Summary
`music_beat_grid(track)`	I	A regular grid of beat times aligned to the first onset.
`music_fft(track, bands)`	I	Spectrum bands at the current playback position.
`music_fft_groups(track, size, groups)`	A	Log-spaced loudness groups from one large FFT at the current playback position.

`music_load(path)`

Decode an audio file into memory and return a track handle.

- **Aliases:** zh `载入音乐` · ja `音楽読込` · ko `음악로드` · th `โหลดเพลง`
- **Parameters:**
- `path` (string) - WAV, FLAC, OGG (Vorbis), MP3 or AAC/M4A.
- **Returns:** track handle (integer from 0), or `-1` on failure (the reason is printed to standard error).
- **Notes:** The whole file is decoded at once, producing an interleaved stereo copy for playback and a mono mix-down for analysis. Long files cost memory proportional to their length.

```
bind start = do {
  bind missing = music_load("no_such_file.wav")
  print(missing)
}
```

```
OUTPUT
-1
```

`music_duration(track)`

Length of a decoded track in seconds.

- **Aliases:** zh `音乐时长` · ja `音楽長さ` · ko `음악길이` · th `ความยาวเพลง`
- **Parameters:**
- `track` (integer) - handle from `music_load`.
- **Returns:** seconds as a number; `0` for an unknown handle.

`music_bpm(track)`

Estimate the tempo of a track in beats per minute.

- **Aliases:** zh `节拍速度` · ja `テンポ` · ko `템포` · th `จังหวะต่อนาที`
- **Parameters:**
- `track` (integer)
- **Returns:** BPM rounded to one decimal place, folded into the range 70 to 170 by doubling or halving; `0` for an unknown handle or a very short track.
- **Notes:** Uses the autocorrelation of the onset envelope between 60 and 200 BPM. It is an estimate: expect it to be off by an octave on some material and by a few tenths on a perfect loop.

`music_key(track)`

Estimate the musical key of a track.

- **Aliases:** zh `调性` · ja `調性` · ko `조성` · th `คีย์เพลง`
- **Parameters:**
- `track` (integer)
- **Returns:** a string such as `"A minor"` or `"C major"`; an empty string for an unknown handle.
- **Notes:** Chroma analysis against Krumhansl-Kessler key profiles. Only major and minor are reported.

`music_onsets(track)`

Times at which new notes or hits begin.

- **Aliases:** zh `音符起点` · ja `オンセット` · ko `온셋` · th `จุดเริ่มเสียง`
- **Parameters:**
- `track` (integer)
- **Returns:** list of times in seconds, in ascending order (empty for an unknown handle).
- **Notes:** Adaptive peak-picking on a spectral-flux envelope. Use it to flash visuals on hits or to build rhythm-game charts.

`music_beat_grid(track)`

A regular grid of beat times aligned to the first onset.

- **Aliases:** zh `节拍网格` · ja `ビートグリッド` · ko `비트그리드` · th `กริดจังหวะ`
- **Parameters:**
- `track` (integer)
- **Returns:** list of beat times in seconds at the detected tempo, starting at the phase of the first onset and continuing to the end of the track.
- **Notes:** The grid is regular. It does not follow tempo changes.

`music_fft(track, bands)`

Spectrum bands at the current playback position.

- **Aliases:** zh `音乐频谱` · ja `音楽スペクトル` · ko `음악스펙트럼` · th `สเปกตรัมเพลง`
- **Parameters:**
- `track` (integer) - the track whose samples are analysed.
- `bands` (integer, default 16) - number of logarithmically spaced bands from 20 Hz to the Nyquist frequency.
- **Returns:** list of `bands` numbers, index 0 being the bass. Values are smoothed magnitudes (not normalised to 1).
- **Notes:** The analysed window is 2048 samples starting at the engine's current position (`music_pos`), so the call returns silence until `music_play` has started. The smoothing state is shared with `mic_fft`.

```
GUI
bind start = do {
  bind t = music_load("song.ogg")
  music_play(t)
  bind bars = music_fft(t, 8)
  print(bars)
}
```

`music_play(track)`

Start playing a track.

- **Aliases:** zh `播放音乐` · ja `音楽再生` · ko `음악재생` · th `เล่นเพลง`
- **Parameters:**
- `track` (integer) - handle from `music_load`.
- **Returns:** nothing.

- **Notes:**
- Always makes `track` the engine's current track and starts it from the beginning, so calling it again restarts the song; there is no separate resume builtin. To resume after `music_pause`, call `music_play` with an id that is not a loaded track (for example `-1`), which leaves the current track alone and just continues it; or note `music_pos()` first and call `music_seek` after `music_play`.
- Playback stops by itself at the end of the track (it does not loop). The music output stream is opened on first use; if that fails a message is printed and the call does nothing.
- On the web target `music_seek` is a no-op and `music_volume` always addresses slot 0.

```
GUI
bind start = do {
  bind song = music_load("song.ogg")
  music_volume(0.8)
  music_play(song)
}
```

`music_pause()`

Pause playback, keeping the position.

- **Aliases:** zh `暂停音乐` · ja `音楽一時停止` · ko `음악일시정지` · th `หยุดเพลงชั่วคราว`
- **Returns:** nothing.
- **Notes:** See `music_play` for how to resume.

```
GUI
bind start = do {
  music_pause()
}
```

`music_stop()`

Stop playback and rewind to the start.

- **Aliases:** zh `停止音乐` · ja `音楽停止` · ko `음악정지` · th `หยุดเพลง`
- **Returns:** nothing.

```
GUI
bind start = do {
  music_stop()
}
```

`music_seek(seconds)`

Jump to a position in the current track.

- **Aliases:** zh `定位音乐` · ja `音楽シーク` · ko `음악탐색` · th `ค้นหาเพลง`
- **Parameters:**
- `seconds` (number, default 0)
- **Returns:** nothing.

```
GUI
bind start = do {
  music_seek(30.0)
}
```

`music_pos()`

Current playback position in seconds.

- **Aliases:** zh `音乐位置` · ja `音楽位置` · ko `음악위치` · th `ตำแหน่งเพลง`
- **Returns:** seconds as a number; `0` before anything has played.

```
GUI
bind start = do {
  print(music_pos())
}
```

`music_volume(level)`

Master volume of the music engine (track and synthesiser together).

- **Aliases:** zh `音乐音量` · ja `音楽音量` · ko `음악음량` · th `ระดับเพลง`
- **Parameters:**

- `level` (number, default 0.8) - clamped to 0..2.
- **Returns:** nothing.
- **Notes:** The output is soft-clipped with `tanh`, so levels above 1 distort gently rather than clip hard.

```
GUI
bind start = do {
  music_volume(0.6)
}
```

Synthesis

The music engine contains a polyphonic synthesiser. An *instrument* is a patch file in the `.ling` patch format: plain `key=value` tokens separated by spaces or newlines, with `#` starting a comment. Operators are numbered from 1 and an operator may frequency-modulate a lower-numbered one.

```
OUTPUT
name = "Rhodes EP"
gm = 5
poly = 16
op1.wave=sine  op1.ratio=1.0  op1.level=1.0  op1.adsr=0.002,1.2,0.0,0.4
op2.wave=sine  op2.ratio=14.0 op2.level=0.5  op2.fm=op1  op2.adsr=0.001,0.2,0.0,0.2
filter.cutoff=0.8 filter.reso=0.2 filter.env=0.3 filter.adsr=0,0.5,0.2,0.3
lfo.rate=5.0 lfo.pitch=0.05
amp = 0.7
```

Recognised keys: `name`, `gm` (General MIDI program number), `poly` (voices, default 16), `amp`, `lfo.rate`, `lfo.pitch`, `lfo.cutoff`, `lfo.amp`, `filter.cutoff`, `filter.reso`, `filter.env`, `filter.adsr`, `filter.on` (any `filter.*` key switches the filter on), and per operator `opN.wave` (`sine`, `saw`, `square` / `pulse`, `tri`, `noise`), `opN.ratio`, `opN.fixed` (fixed frequency in Hz), `opN.level`, `opN.detune`, `opN.adsr` (attack, decay, sustain, release, comma separated) and `opN.fm` (the operator it modulates). Unknown keys are ignored. The repository ships ready-made patches in `instruments/`.

Builtin	Level	Summary
`music_patch(path)`	B	Load an instrument patch and return its handle.

Builtin	Level	Summary
`music_note(instrument, pitch, duration, velocity)`	B	Play a note of fixed length.
`music_note_on(instrument, pitch, velocity)`	I	Press a key and hold it.
`music_note_off(instrument, pitch)`	I	Release a key pressed with `music_note_on`.

`music_patch(path)`

Load an instrument patch and return its handle.

- **Aliases:** zh `乐器音色` · ja `音色読込` · ko `악기패치` · th `แพตช์เครื่องดนตรี`
- **Parameters:**
- `path` (string) - a `.ling` patch file.
- **Returns:** instrument handle (integer from 0), or `-1` if the file cannot be read or the music engine could not start.
- **Notes:** Patch handles are passed as the first argument of `music_note`, `music_note_on` and `music_note_off`. They are unrelated to track handles.

```
GUI
bind start = do {
  bind piano = music_patch("instruments/grand_piano.ling")
  music_note(piano, "C4", 0.5, 0.9)
}
```

`music_note(instrument, pitch, duration, velocity)`

Play a note of fixed length.

- **Aliases:** zh `弹音符` · ja `音符演奏` · ko `음표연주` · th `เล่นโน้ต`
- **Parameters:**
- `instrument` (integer, default 0) - handle from `music_patch`.
- `pitch` (number or string, default 60) - MIDI number or note name.
- `duration` (number, default 0.5) - seconds the key is held; the patch's release tail follows.
- `velocity` (number, default 0.9) - 0 to 1.
- **Returns:** nothing.

```
GUI
bind start = do {
  bind lead = music_patch("instruments/square_lead.ling")
  music_note(lead, "E5", 0.25, 0.8)
}
```

`music_note_on(instrument, pitch, velocity)`

Press a key and hold it.

- **Aliases:** zh `音符开始` · ja `音符オン` · ko `음표켜기` · th `โน้ตเริ่ม`
- **Parameters:**
- `instrument` (integer, default 0)
- `pitch` (number or string, default 60)
- `velocity` (number, default 0.9)
- **Returns:** nothing.
- **Notes:** Pair every `music_note_on` with a `music_note_off` of the same instrument and pitch.

```
GUI
bind start = do {
  bind organ = music_patch("instruments/drawbar_organ.ling")
  music_note_on(organ, 48, 0.7)
}
```

`music_note_off(instrument, pitch)`

Release a key pressed with `music_note_on`.

- **Aliases:** zh `音符结束` · ja `音符オフ` · ko `음표끄기` · th `โน้ตจบ`
- **Parameters:**
- `instrument` (integer, default 0)
- `pitch` (number or string, default 60)
- **Returns:** nothing.

```
GUI
bind start = do {
  music_note_off(0, 48)
}
```

Pitch and note names

Builtin	Level	Summary
`music_hz(pitch)`	B	Convert a MIDI number or note name to a frequency.
`music_note_name(hz)`	B	Name of the nearest note to a frequency.
`music_pitch_score(sung_hz, target_hz)`	I	Score a sung pitch against a target.
`music_mic_pitch()`	I	Fundamental frequency heard on the microphone.

`music_hz(pitch)`

Convert a MIDI number or note name to a frequency.

- **Aliases:** zh `音符频率` · ja `音符周波数` · ko `음표주파수` · th `ความถี่โน้ต`
- **Parameters:**
- `pitch` (number or string, default 69) - MIDI number or note name. An unparsable string is read as MIDI 69.
- **Returns:** frequency in Hz. A4 (69) is 440.

```
bind start = do {
  print(music_hz("A4"))
  print(round(music_hz(60) * 100) / 100)
  print(round(music_hz("C5") * 100) / 100)
}
```

```
OUTPUT
440
261.63
523.25
```

`music_note_name(hz)`

Name of the nearest note to a frequency.

- **Aliases:** zh `音名` · ja `音名称` · ko `음이름` · th `ชื่อโน้ต`
- **Parameters:**
- `hz` (number, default 0)
- **Returns:** a string such as `"A4"` or `"C#5"` (sharp spelling); an empty string if `hz` is not positive.

```
bind start = do {
  print(music_note_name(440.0))
  print(music_note_name(262.0))
  print(music_note_name(0.0))
}
```

```
OUTPUT
A4
C4
```

`music_pitch_score(sung_hz, target_hz)`

Score a sung pitch against a target.

- **Aliases:** zh `音准评分` · ja `音程スコア` · ko `음정점수` · th `คะแนนเสียง`
- **Parameters:**
- `sung_hz` (number) - detected pitch, for example from `music_mic_pitch`.
- `target_hz` (number)
- **Returns:** a number from 0 to 1: 1 is exactly on pitch, 0 is a semitone or more off. The error is folded to the nearest octave, so singing an octave away still scores well. If either argument is not positive the result is 0.

```
bind start = do {
  print(music_pitch_score(440.0, 440.0))
  print(round(music_pitch_score(450.0, 440.0) * 100) / 100)
  print(music_pitch_score(0.0, 440.0))
}
```

```
OUTPUT
1
0.61
0
```

`music_mic_pitch()`

Fundamental frequency heard on the microphone.

- **Aliases:** zh `麦克风音高` · ja `マイク音程` · ko `마이크음정` · th `ระดับเสียงไมค์`
- **Returns:** Hz as a number, or `0` when the microphone is not open, the signal is too quiet or no clear pitch is found.
- **Notes:** The microphone must have been opened with `mic_open` first. Detection searches 55 to 1000 Hz with a normalised autocorrelation, which is suited to a single voice or lead instrument.

```
GUI
bind start = do {
  mic_open()
  print(music_mic_pitch())
}
```

Rhythm-game judging

Timing windows, as the absolute timing error in milliseconds: Perfect up to 30, Great up to 60, Good up to 100, Ok up to 140, anything later is a Miss.

Builtin	Level	Summary
`music_judge(delta_ms)`	I	Grade a hit by its timing error.
`music_grade_name(index)`	I	Display name of a grade index.

`music_judge(delta_ms)`

Grade a hit by its timing error.

- **Aliases:** zh `判定` · ja `判定する` · ko `판정` · th `ตัดสินจังหวะ`

- **Parameters:**
- `delta_ms` (number, default 9999) - signed error in milliseconds between the key press and the note time; the sign is ignored.
- **Returns:** grade index: 0 Perfect, 1 Great, 2 Good, 3 Ok, 4 Miss.

```
bind start = do {
  print(music_judge(12.0))
  print(music_judge(0.0 - 45.0))
  print(music_judge(90.0))
  print(music_judge(500.0))
}
```

```
OUTPUT
0
1
2
4
```

`music_grade_name(index)`

Display name of a grade index.

- **Aliases:** zh `判定名` · ja `判定名称` · ko `판정이름` · th `ชื่อการตัดสิน`
- **Parameters:**
- `index` (integer, default 4) - a value returned by `music_judge`. Out-of-range values give the Miss name.
- **Returns:** `"PERFECT"`, `"GREAT"`, `"GOOD"`, `"OK"` or `"MISS"`.

```
bind start = do {
  print(music_grade_name(music_judge(45.0)))
  print(music_grade_name(4))
}
```

```
OUTPUT
GREAT
MISS
```

Lyrics (.lrc) and MIDI files

Builtin	Level	Summary
`music_lrc(path)`	I	Load a timed-lyrics file in `.lrc` format.
`music_lyric(lyrics, time)`	I	The lyric line active at a given time.
`music_midi_load(path)`	I	Load a Standard MIDI File as a silent note source.
`music_midi_count(midi)`	I	Number of notes in a loaded MIDI file.
`music_midi_notes(midi)`	I	Note onsets of a MIDI file as a flat list.
`music_midi_bars(midi)`	I	Notes of a MIDI file with durations, for drawing note bars.

`music_lrc(path)`

Load a timed-lyrics file in `.lrc` format.

- **Aliases:** zh `载入歌词` · ja `歌詞読込` · ko `가사로드` · th `โหลดเนื้อเพลง`
- **Parameters:**
- `path` (string) - a text file whose lines look like `[mm:ss.xx] text`. A line may carry several time stamps; malformed lines are ignored.
- **Returns:** lyrics handle (integer from 0), or `-1` if the file cannot be read.

```
bind start = do {
  write_file("song.lrc", "[00:01.00] First line\n[00:03.50] Second line\n[00:06.00]
Third line\n")
  bind lyrics = music_lrc("song.lrc")
  print(lyrics)
  print(music_lyric(lyrics, 4.0))
}
```

```
OUTPUT
0
Second line
```

`music_lyric(lyrics, time)`

The lyric line active at a given time.

- **Aliases:** zh `当前歌词` · ja `現在歌詞` · ko `현재가사` · th `เนื้อเพลงปัจจุบัน`
- **Parameters:**
- `lyrics` (integer) - handle from `music_lrc`.
- `time` (number, default 0) - seconds; usually `music_pos()`.
- **Returns:** the text of the latest line whose time stamp is not after `time`; an empty string before the first line.

```
bind start = do {
  write_file("song.lrc", "[00:01.00] First line\n[00:03.50] Second line\n")
  bind lyrics = music_lrc("song.lrc")
  print("[" + music_lyric(lyrics, 0.5) + "]")
  print(music_lyric(lyrics, 2.0))
  print(music_lyric(lyrics, 99.0))
}
```

```
OUTPUT
[]
First line
Second line
```

`music_midi_load(path)`

Load a Standard MIDI File as a silent note source.

- **Aliases:** zh `载入MIDI` · ja `MIDI読込` · ko `미디로드` · th `โหลดมิดี`
- **Parameters:**
- `path` (string) - a `.mid` file.
- **Returns:** MIDI handle (integer from 0), or `-1` on failure.
- **Notes:** Loading does not play anything. The notes are data you can use to spawn coins, show falling bars or trigger cues.

```ling
bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind t = daw_track_add("synth")
  bind c = daw_clip_add_midi(t, 0, 4)
  daw_note_add(t, c, 0, 1, 60, 0.8)
  daw_note_add(t, c, 1, 1, 64, 0.8)
  daw_note_add(t, c, 2, 2, 67, 0.8)
  daw_midi_export_project("song.mid")

  bind m = music_midi_load("song.mid")
  print(m)
  print(music_midi_count(m))
}
```

```
OUTPUT
0
3
```

`music_midi_count(midi)`

Number of notes in a loaded MIDI file.

- **Aliases:** zh `MIDI数量` · ja `MIDI数` · ko `미디수` · th `จำนวนมิดี`
- **Parameters:**
- `midi` (integer) - handle from `music_midi_load`.
- **Returns:** note count; `0` for an unknown handle.

`music_midi_notes(midi)`

Note onsets of a MIDI file as a flat list.

- **Aliases:** zh `MIDI音符` · ja `MIDIノート` · ko `미디음표` · th `โน้ตมิดี`
- **Parameters:**
- `midi` (integer)
- **Returns:** flat list `[time, midi, time, midi, ...]` with two entries per note: onset time in seconds, then the MIDI note number.

```
bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind t = daw_track_add("synth")
  bind c = daw_clip_add_midi(t, 0, 4)
  daw_note_add(t, c, 0, 1, 60, 0.8)
  daw_note_add(t, c, 1, 1, 64, 0.8)
  daw_note_add(t, c, 2, 2, 67, 0.8)
  daw_midi_export_project("song.mid")

  bind m = music_midi_load("song.mid")
  print(music_midi_notes(m))
}
```

```
OUTPUT
[0, 60, 0.5, 64, 1, 67]
```

`music_midi_bars(midi)`

Notes of a MIDI file with durations, for drawing note bars.

- **Aliases:** zh `MIDI音条` · ja `MIDIバー` · ko `미디바` · th `แท่งมิดี`
- **Parameters:**
- `midi` (integer)
- **Returns:** flat list `[time, midi, duration, ...]` with three entries per note, times and durations in seconds.

```
bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind t = daw_track_add("synth")
  bind c = daw_clip_add_midi(t, 0, 4)
  daw_note_add(t, c, 0, 1, 60, 0.8)
  daw_note_add(t, c, 2, 2, 67, 0.8)
  daw_midi_export_project("song.mid")

  bind m = music_midi_load("song.mid")
  print(music_midi_bars(m))
}
```

```
OUTPUT
[0, 60, 0.5, 1, 67, 1]
```

The DAW builtins

The `daw_*` builtins drive the engine of LingStationX, the digital audio workstation that ships with Ling. The same engine is available to ordinary scripts, so you can build projects, edit notes, run effects and render audio without opening any window. The engine lives in the `ling-daw` crate; the interpreter creates one instance the first time a `daw_*`, `fs_*` or `dlg_*` builtin is called.

Things to know before using it:

- **Beats, not seconds.** Clip positions, lengths, loop points and render ranges are in *beats*. Convert with `daw_secs_at(beat)` or the tempo.
- **Indices.** Tracks, clips, notes and effects are addressed by zero-based index. Indices shift when you remove or insert items, so re-read counts after structural edits.
- **Return values.** Booleans come back as the numbers `1` and `0`. Functions that create something return its index, or `-1` on failure.
- **No audio device needed for offline work.** Editing, `daw_bounce` and `daw_render_start` run without an output device. `daw_play` starts real-time playback and is audible when a device exists, so avoid it in automated scripts.
- **Status text.** Many operations leave a human-readable message; read it once with `daw_status`.

- **Tracks and instruments.** `daw_track_add` accepts the kinds `"synth"`, `"drums"`, `"audio"`, `"sf2"` (with a soundfont path) and `"growl"`, `"voice"`, `"prism"`, `"sampler"`, `"pluck"`, `"chip"`, `"cinema"`, `"script"`. An unknown kind creates no track: the interpreter (`ling run --interp`) stops with a runtime error, while the default JIT backend lets the call yield unit and the script continues.
- **Effect targets.** Functions that take a *target* operate on a track when it is a track index and on the master bus when it is `-1`.
- **Families not documented entry by entry.** The engine has several hundred further `daw_*` builtins for song generation, mixing, vocals and export. They are listed by family at the end of this chapter.

A complete offline session:

Build, process and render a song

```
bind start = do {
  daw_new()
  daw_set_project_name("Demo")
  daw_set_bpm(100)

  bind lead = daw_track_add("synth")
  daw_inst_preset(lead, "FM Bell")
  bind clip = daw_clip_add_midi(lead, 0, 4)
  bind pitches = [60, 64, 67, 72]
  bind i = 0
  while i < 4 {
    daw_note_add(lead, clip, i, 0.75, pitches[i], 0.8)
    bind i = i + 1
  }

  bind verb = daw_fx_add(lead, "Reverb")
  daw_fx_set_wet(lead, verb, 0.3)

  print("tracks: " + str(daw_track_count()))
  print("notes: " + str(len(daw_clip_notes(lead, clip)) / 7))
  print("project ends at beat " + str(daw_end_beat()))
  print("render ok: " + str(daw_bounce("demo.wav", 0, 4, 1.0, 16, "wav", 0.5)))
}
```

```
OUTPUT
tracks: 1
notes: 4
project ends at beat 4
render ok: 1
```

Session and project

Builtin	Level	Summary
`daw_sample_rate()`	A	Sample rate of the engine in Hz.
`daw_device_rate()`	A	Native rate of the audio output device in Hz.
`daw_device()`	A	Name of the audio output device in use.
`daw_devices()`	A	Names of all audio output devices.
`daw_status()`	I	Take the last status message.
`daw_new()`	I	Start an empty project.
`daw_project_name()`	I	Name of the project.
`daw_set_project_name(name)`	I	Rename the project.
`daw_project_path()`	I	File the project was last saved to or loaded from.
`daw_set_project_path(path)`	A	Set the project path without saving.
`daw_project_save(path)`	I	Save the project, with its audio media, to a file.
`daw_project_load(path)`	I	Load a project file.
`daw_project_save_copy(path)`	A	Save a copy without changing the project path.
`daw_meta_get(key)`	A	Read a project metadata value.
`daw_meta_set(key, value)`	A	Store a project metadata value.
`daw_dirty()`	A	Whether the project has unsaved changes.
`daw_set_dirty(flag)`	A	Set or clear the unsaved-changes flag.

`daw_status()`

Take the last status message.

- **Returns:** the most recent message, for example `"Rendered demo.wav"` or `"Open failed: ..."`. Reading it clears it, so a second call returns an empty string.

```
bind start = do {
  daw_new()
  daw_bounce("status_demo.wav", 0, 1, 0.0, 16, "wav", 0.5)
  print(daw_status())
  print("[" + daw_status() + "]")
}
```

```
OUTPUT
Rendered status_demo.wav
[]
```

`daw_new()`

Start an empty project.

- **Returns:** nothing.
- **Notes:** Replaces the current project with one named "Untitled" at 120 BPM, with a master compressor, and clears undo history and the project path. There is no confirmation.

```
bind start = do {
  daw_new()
  print(daw_project_name())
  print(daw_bpm())
  print(daw_track_count())
}
```

```
OUTPUT
Untitled
120
0
```

`daw_project_name()`

Name of the project.

- **Returns:** string.

`daw_set_project_name(name)`

Rename the project.

- **Parameters:**
- `name` (string)
- **Returns:** nothing.

`daw_project_path()`

File the project was last saved to or loaded from.

- **Returns:** string; empty for a new project.

`daw_project_save(path)`

Save the project, with its audio media, to a file.

- **Parameters:**
- `path` (string, default: the current project path) - destination. An empty string with no current path fails.
- **Returns:** `1` on success, `0` on failure (see `daw_status`). Success also sets the project path and clears the dirty flag.

```
bind start = do {
  daw_new()
  daw_set_project_name("Saved")
  print(daw_project_save("saved_project.lsx"))
  print(daw_project_path())
}
```

```
OUTPUT
1
saved_project.lsx
```

`daw_project_load(path)`

Load a project file.

- **Parameters:**
- `path` (string)
- **Returns:** `1` on success, `0` on failure.
- **Notes:** Resets undo history.

```
bind start = do {
  daw_new()
  daw_set_project_name("Roundtrip")
  daw_set_bpm(90)
  daw_project_save("roundtrip.lsx")
  daw_new()
  print(daw_project_name())
  print(daw_project_load("roundtrip.lsx"))
  print(daw_project_name())
  print(daw_bpm())
}
```

```
OUTPUT
Untitled
1
Roundtrip
90
```

`daw_play()`

Start real-time playback from the current position.

- **Returns:** nothing.
- **Notes:** Audible when an output device is open. For offline rendering use `daw_bounce`.

`daw_pause()`

Pause playback.

- **Returns:** nothing.

`daw_stop()`

Stop playback and return to the start.

- **Returns:** nothing.

`daw_seek(beat)`

Move the play position.

- **Parameters:**
- `beat` (number)
- **Returns:** nothing.

`daw_playing()`

Whether the transport is running.

- **Returns:** `1` or `0`.

`daw_position()`

Current play position.

- **Returns:** beats as a number.

`daw_bpm()`

Project tempo.

- **Returns:** BPM as a number.

`daw_set_bpm(bpm)`

Set the project tempo.

- **Parameters:**
- `bpm` (number) - clamped to 20..400.
- **Returns:** nothing.

```
bind start = do {
  daw_new()
  daw_set_bpm(1000)
  print(daw_bpm())
  daw_set_bpm(96)
  print(daw_bpm())
}
```

```
OUTPUT
400
96
```

`daw_end_beat()`

Beat at which the last clip ends.

- **Returns:** beats as a number; `0` for an empty project.

`daw_master_get(what)`

Read a master-bus value.

- **Parameters:**
- `what` (string) - `"vol"`, `"meter_l"` or `"meter_r"`. Anything else returns 0.
- **Returns:** number. Meters hold the latest output level.

`daw_master_set(what, value)`

Set a master-bus value.

- **Parameters:**
- `what` (string) - only `"vol"` is accepted.
- `value` (number) - clamped to 0..2.
- **Returns:** nothing.

```
bind start = do {
  daw_new()
  daw_master_set("vol", 0.5)
  print(daw_master_get("vol"))
}
```

```
OUTPUT
0.5
```

`daw_track_count()`

Number of tracks.

- **Returns:** integer.

`daw_track_add(kind, path)`

Add a track.

- **Parameters:**
- `kind` (string) - `"synth"`, `"drums"`, `"audio"`, `"sf2"`, `"growl"`, `"voice"`, `"prism"`, `"sampler"`, `"pluck"`, `"chip"`, `"cinema"` or `"script"`.
- `path` (string, optional) - for instrument kinds, a patch file to load; for `"sf2"`, the SoundFont file (required).
- **Returns:** index of the new track, or `-1` if the track limit is reached or a SoundFont cannot be loaded.
- **Notes:**
- New tracks start with a gentle compressor in their effect chain. Colours cycle through a seven-colour palette.
- An unknown `kind` is an error, not a `-1` return: under `ling run --interp` the script stops with `unknown instrument type`; under the default backend the call yields unit and no track is created.

```
bind start = do {
  daw_new()
  print(daw_track_add("synth"))
  print(daw_track_add("drums"))
  print(daw_track_add("audio"))
  print(daw_track_count())
}
```

```
OUTPUT
0
1
2
3
```

`daw_track_remove(track)`

Delete a track and its clips.

- **Parameters:**
- `track` (integer)
- **Returns:** nothing.

`daw_track_name(track)`

Name of a track.

- **Parameters:**
- `track` (integer)
- **Returns:** string; empty for an invalid index.

`daw_track_set_name(track, name)`

Rename a track.

- **Parameters:**
- `track` (integer)
- `name` (string)

- **Returns:** nothing.

`daw_track_kind(track)`

Instrument kind of a track.

- **Parameters:**
- `track` (integer)
- **Returns:** `"audio"` for audio tracks, otherwise the instrument kind such as `"synth"` or `"drums"`.

`daw_track_get(track, field)`

Read a track property.

- **Parameters:**
- `track` (integer)
- `field` (string) - `"vol"`, `"pan"`, `"mute"`, `"solo"`, `"meter_l"`, `"meter_r"`, `"asleep"`, `"audio"` (1 for an audio track), `"cr"`, `"cg"`, `"cb"` (track colour channels), `"clips"` (clip count) or `"fx"` (effect count).
- **Returns:** number; `0` for an unknown field or invalid track.

`daw_track_set(track, field, value)`

Change a track property.

- **Parameters:**
- `track` (integer)
- `field` (string) - `"vol"` (clamped 0..2), `"pan"` (-1..1), `"mute"` (above 0.5 means on), `"solo"`, or `"color"` (palette index 0..6).
- `value` (number)
- **Returns:** nothing.

```
bind start = do {
  daw_new()
  bind t = daw_track_add("synth")
  daw_track_set(t, "vol", 0.5)
  daw_track_set(t, "pan", -0.25)
  daw_track_set(t, "mute", 1)
  print(daw_track_get(t, "vol"))
  print(daw_track_get(t, "pan"))
  print(daw_track_get(t, "mute"))
}
```

```
OUTPUT
0.5
-0.25
1
```

`daw_clip_count(track)`

Number of clips on a track.

- **Parameters:**
- `track` (integer)
- **Returns:** integer.

`daw_clip_add_midi(track, start, length)`

Add an empty MIDI clip.

- **Parameters:**
- `track` (integer)
- `start` (number) - beats, at least 0.
- `length` (number) - beats, at least 0.25.
- **Returns:** clip index, or `-1` for an invalid track.

`daw_clip_add_audio(track, path, start)`

Add an audio clip from a file.

- **Parameters:**
- `track` (integer)
- `path` (string) - audio file to decode.
- `start` (number) - beats.
- **Returns:** clip index, or `-1` if the file cannot be decoded (see `daw_status`). The clip length is derived from the file duration and the tempo.

`daw_clip_get(track, clip, field)`

Read a clip property.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `field` (string) - `"start"`, `"len"`, `"loop"`, `"end"` (all beats); `"audio"` (1 for audio clips); `"notes"` (note count, MIDI clips); for audio clips `"gain"`, `"pitch"` (semitones), `"warp"` (target BPM, 0 for off), `"offset"` and `"seconds"` (source length).
- **Returns:** number; `0` for unknown fields or invalid indices.

`daw_clip_set(track, clip, field, value)`

Change a clip property.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `field` (string) - `"start"`, `"len"` (at least 0.25), `"loop"` (loop length in beats; below 0.25 turns looping off); audio clips also `"gain"` (0..4), `"pitch"` (-24..24 semitones), `"warp"` (0 for off, otherwise 20..400 BPM) and `"offset"`.
- `value` (number)
- **Returns:** nothing.

`daw_clip_remove(track, clip)`

Delete a clip.

- **Parameters:**
- `track` (integer)

- `clip` (integer)
- **Returns:** nothing.

`daw_clip_dup(track, clip, start)`

Copy a clip to a new start position on the same track.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `start` (number) - beats.
- **Returns:** index of the copy, or `-1`.

`daw_clip_split(track, clip, beat)`

Cut a clip in two.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `beat` (number) - absolute beat of the cut; it must lie strictly inside the clip.
- **Returns:** index of the right-hand half, or `-1` if the cut is outside the clip.

`daw_clip_notes(track, clip)`

All notes of a MIDI clip.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- **Returns:** flat list with seven numbers per note (see above); empty for audio clips.

```
bind start = do {
  daw_new()
  bind t = daw_track_add("synth")
  bind c = daw_clip_add_midi(t, 0, 4)
  daw_note_add(t, c, 0, 1, 60, 0.5)
  daw_note_add(t, c, 1, 2, 64, 1.0)
  print(daw_clip_notes(t, c))
}
```

```
OUTPUT
[0, 1, 60, 0.5, 0, 0.5, 0.5, 1, 2, 64, 1, 0, 0.5, 0.5]
```

`daw_clip_set_notes(track, clip, notes)`

Replace every note of a MIDI clip at once.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `notes` (list) - flat list read in groups of seven numbers per note (`start, length, key, velocity, pan, cutoff, resonance`). Only the last note may be given with just its first four numbers. Keys are rounded and clamped to 0..127, velocity to 0.01..1.
- **Returns:** `1` on success, `0` if the clip is not a MIDI clip.

```
bind start = do {
  daw_new()
  bind t = daw_track_add("synth")
  bind c = daw_clip_add_midi(t, 0, 4)
  daw_clip_set_notes(t, c, [0, 1, 60, 0.8, 0, 0.5, 0.5, 1, 2, 64, 0.8, 0, 0.5, 0.5])
  print(daw_clip_get(t, c, "notes"))
}
```

```
OUTPUT
2
```

`daw_note_add(track, clip, start, length, key, velocity)`

Add a note to a MIDI clip.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `start` (number) - beats from the clip start.
- `length` (number) - beats, at least 0.03.
- `key` (number) - MIDI note number, clamped to 0..127.
- `velocity` (number) - clamped to 0.01..1.
- **Returns:** note index within the clip, or `-1` if the clip is not a MIDI clip.
- **Notes:** For a drum track, the key selects the drum voice (36 kick, 38 snare, 42 closed hat, …).

`daw_note_set(track, clip, note, field, value)`

Change one property of a note.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `note` (integer) - index from `daw_note_add` or order in `daw_clip_notes`.
- `field` (string) - `"start"`, `"len"`, `"key"`, `"vel"`, `"pan"`, `"cutoff"` or `"reso"`.
- `value` (number)
- **Returns:** nothing.

`daw_note_remove(track, clip, note)`

Delete a note.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `note` (integer)
- **Returns:** nothing. Later notes shift down by one index.

Instruments

Builtin	Level	Summary
`daw_inst_presets(track)`	I	Preset names of the track's instrument.
`daw_inst_preset(track, name)`	I	Load a preset by name.
`daw_inst_params(track)`	I	All parameters of the instrument.
`daw_inst_get(track, name)`	I	Read one instrument parameter.
`daw_inst_set(track, name, value)`	I	Set one instrument parameter.
`daw_live(track, key, velocity, on)`	A	Play or release a note on a track's instrument right now.

`daw_inst_presets(track)`

Preset names of the track's instrument.

- **Parameters:**
- `track` (integer)
- **Returns:** list of strings; empty for audio tracks.

```
bind start = do {
  daw_new()
  bind t = daw_track_add("synth")
  print(daw_inst_presets(t))
}
```

```
OUTPUT
[Init Saw, Sub Bass, Additive Organ, FM Bell, Hybrid Pad]
```

`daw_inst_preset(track, name)`

Load a preset by name.

- **Parameters:**
- `track` (integer)
- `name` (string) - one of the names from `daw_inst_presets`.
- **Returns:** nothing.

`daw_inst_params(track)`

All parameters of the instrument.

- **Parameters:**
- `track` (integer)
- **Returns:** flat list `[name, value, name, value, ...]`.

`daw_inst_get(track, name)`

Read one instrument parameter.

- **Parameters:**
- `track` (integer)
- `name` (string) - a name from `daw_inst_params`.
- **Returns:** the value, or `-1` if the instrument has no such parameter.

`daw_inst_set(track, name, value)`

Set one instrument parameter.

- **Parameters:**
- `track` (integer)
- `name` (string)
- `value` (number)
- **Returns:** nothing.

`daw_fx_types()`

Names of all available effects.

- **Returns:** list of strings.

```
bind start = do {
  print(daw_fx_types())
}
```

```
OUTPUT
[EQ, Compressor, Drive, Delay, Reverb, Chorus, Vox Filter, Vox Reverb, Vocal Halo,
Rainbow Hypergrowl, Ducker, Vocoder, Ringstorm, OTT, Split Drive, Comb, Phaser,
Flanger, Freq Shifter, Imager, Transient, Articulator, Colour Bass, Param EQ, Volume
Shaper, Hyper Chorus, Stutter, Limiter]
```

`daw_fx_count(target)`

Number of effects in a chain.

- **Parameters:**
- `target` (integer) - track index or `-1` for the master.
- **Returns:** integer.

`daw_fx_name(target, index)`

Name of an effect in a chain.

- **Parameters:**
- `target` (integer)
- `index` (integer)
- **Returns:** string.

`daw_fx_add(target, name)`

Append an effect to a chain.

- **Parameters:**
- `target` (integer)
- `name` (string) - one of the names from `daw_fx_types`.
- **Returns:** index of the new effect, or `-1` for an unknown name or target.

```
bind start = do {
  daw_new()
  bind t = daw_track_add("synth")
  print(daw_fx_count(t))
  bind r = daw_fx_add(t, "Reverb")
  print(r)
  print(daw_fx_name(t, r))
  print(daw_fx_params(t, r))
}
```

```
OUTPUT
1
1
Reverb
[mix, 0.25, size, 0.6000000238418579, damp, 0.4000000059604645]
```

`daw_fx_remove(target, index)`

Delete an effect.

- **Parameters:**
- `target` (integer)
- `index` (integer)
- **Returns:** nothing.

`daw_fx_params(target, index)`

Parameter names and current values of an effect.

- **Parameters:**
- `target` (integer)
- `index` (integer)
- **Returns:** flat list `[name, value, name, value, ...]`; the position of a name in the list (counting pairs from 0) is the `param_index` for `daw_fx_set`.

`daw_fx_set(target, index, param_index, value)`

Set an effect parameter.

- **Parameters:**
- `target` (integer)
- `index` (integer)
- `param_index` (integer) - which parameter, in `daw_fx_params` order.
- `value` (number) - normally 0..1.
- **Returns:** nothing.

`daw_fx_set_wet(target, index, wet)`

Set the dry/wet balance of an effect.

- **Parameters:**
- `target` (integer)
- `index` (integer)
- `wet` (number) - clamped to 0..1.
- **Returns:** nothing.

`daw_bounce(path, start, end, tail, bits, format, quality)`

Render the project to an audio file, synchronously.

- **Parameters:**
- `path` (string) - output file; missing folders are created.
- `start` (number, beats)
- `end` (number, beats) - if not greater than `start`, the end of the project is used.
- `tail` (number, seconds, default 0) - extra time after `end` for reverb and release tails.
- `bits` (integer, default 16) - sample depth for WAV and FLAC.
- `format` (string) - `"wav"`, `"flac"` or `"ogg"`; empty infers it from the file extension and falls back to WAV.
- `quality` (number 0..1, default 0.5) - Ogg Vorbis quality.
- **Returns:** `1` on success, `0` on failure (see `daw_status`).
- **Notes:** Rendering is faster than real time and needs no audio device. For long projects use the asynchronous `daw_render_start` so a window can keep updating.

`daw_midi_import(path, beat)`

Import a song file as new tracks.

- **Parameters:**
- `path` (string) - a MIDI file. The importer also recognises tracker modules and FL Studio projects by their contents.
- `beat` (number) - where the imported material starts.
- **Returns:** number of tracks created. Each MIDI track or channel gets its own track, General MIDI programs map to synth patches and channel 10 becomes a drum kit.

`daw_midi_export(track, path)`

Export one track as a MIDI file.

- **Parameters:**
- `track` (integer)
- `path` (string)
- **Returns:** `1` on success, `0` on failure.

`daw_midi_export_project(path)`

Export the whole project as a MIDI file.

- **Parameters:**
- `path` (string)
- **Returns:** `1` on success, `0` on failure.

`fs_list(folder)`

List a folder.

- **Parameters:**
- `folder` (string)
- **Returns:** flat list `[name, is_folder, size_in_bytes, ...]`, folders first, each group sorted case-insensitively. Names that start with a dot are skipped. An unreadable folder gives an empty list.

`fs_is_dir(path)`

Whether a path is a folder.

- **Parameters:**
- `path` (string)
- **Returns:** `1` or `0`.

`fs_parent(path)`

Parent folder of a path.

- **Parameters:**
- `path` (string)
- **Returns:** string; empty if there is none.

`fs_join(folder, name)`

Join two path parts using the platform separator.

- **Parameters:**
- `folder` (string)
- `name` (string)
- **Returns:** string.

`fs_home()`

The user's home folder.

- **Returns:** string, from `USERPROFILE` or `HOME`.

`fs_cwd()`

The current working directory.

- **Returns:** string.

`fs_mkdir(path)`

Create a folder and any missing parents.

- **Parameters:**
- `path` (string)
- **Returns:** `1` on success, `0` on failure.

`fs_copy(from, to)`

Copy a file.

- **Parameters:**
- `from` (string)
- `to` (string)
- **Returns:** `1` on success, `0` on failure.

`fs_delete(path)`

Delete a file or a whole folder tree.

- **Parameters:**
- `path` (string)
- **Returns:** `1` on success, `0` on failure.
- **Notes:** Folders are removed recursively without confirmation.

`dlg_open(title, extensions, folder)`

Native "open file" dialog.

- **Parameters:**
- `title` (string)
- `extensions` (string) - comma-separated extensions without dots, such as `"wav,flac"`; empty for all files.
- `folder` (string, optional) - folder to start in.
- **Returns:** the chosen path, or an empty string if cancelled.
- **Notes:** Blocks until the dialog closes.

`dlg_save(title, extensions, name, folder)`

Native "save file" dialog.

- **Parameters:**

- `title` (string)
- `extensions` (string) - comma-separated; if the user types no extension, the first one is appended.
- `name` (string) - suggested file name.
- `folder` (string, optional)
- **Returns:** the chosen path, or an empty string if cancelled.

`dlg_folder(title, folder)`

Native "choose folder" dialog.

- **Parameters:**
- `title` (string)
- `folder` (string, optional) - folder to start in.
- **Returns:** the chosen folder, or an empty string if cancelled.

Shared state: the st_ store

LingStationX is written as many small top-level functions. Functions cannot see each other's local variables, so the DAW ships a global key/value store. It is a plain dictionary of Ling values that lives for the whole run and is available to any script.

Builtin	Level	Summary
`st_set(key, value)`	I	Store a value under a key.
`st_get(key, default)`	I	Read a stored value.
`st_has(key)`	I	Whether a key exists.
`st_del(key)`	I	Remove a key.
`st_add(key, amount)`	I	Add to a numeric value and return the result.
`st_lpush(key, value)`	I	Append to a stored list.
`st_llen(key)`	I	Length of a stored list.
`st_lget(key, index)`	I	Element of a stored list.
`st_lset(key, index, value)`	I	Set an element of a stored list, growing it if needed.
`st_ldel(key, index)`	I	Remove one element from a stored list.
`st_lclear(key)`	I	Replace a stored value with an empty list.

`st_set(key, value)`

Store a value under a key.

- **Parameters:**
- `key` (string) - converted to text if it is not already.
- `value` (any) - number, string, list or any other value. Missing means unit.
- **Returns:** nothing.

```
bind start = do {
  st_set("player", "Ling")
  st_set("score", 10)
  print(st_get("player"))
  print(st_get("score"))
}
```

```
OUTPUT
Ling
10
```

`st_get(key, default)`

Read a stored value.

- **Parameters:**
- `key` (string)
- `default` (any, optional) - returned when the key is missing.
- **Returns:** the stored value; otherwise `default`; otherwise the number `0`.

```
bind start = do {
  print(st_get("missing"))
  print(st_get("missing", "none"))
}
```

```
OUTPUT
0
none
```

`st_has(key)`

Whether a key exists.

- **Parameters:**
- `key` (string)
- **Returns:** `1` or `0`.

`st_del(key)`

Remove a key.

- **Parameters:**
- `key` (string)
- **Returns:** `1` if the key was present, `0` otherwise. Use it in long-running programs so the store does not grow without bound.

```
bind start = do {
  st_set("temp", 1)
  print(st_has("temp"))
  print(st_del("temp"))
  print(st_has("temp"))
}
```

```
OUTPUT
1
1
0
```

`st_add(key, amount)`

Add to a numeric value and return the result.

- **Parameters:**
- `key` (string)
- `amount` (number, default 0)
- **Returns:** the new value. A missing or non-numeric entry counts as 0.

```
bind start = do {
  st_add("hits", 1)
  st_add("hits", 1)
  print(st_add("hits", 3))
}
```

```
OUTPUT
5
```

`st_lpush(key, value)`

Append to a stored list.

- **Parameters:**
- `key` (string)
- `value` (any)
- **Returns:** nothing. The list is created on first use.

`st_llen(key)`

Length of a stored list.

- **Parameters:**
- `key` (string)
- **Returns:** number of elements; `0` if the key is missing or not a list.

`st_lget(key, index)`

Element of a stored list.

- **Parameters:**
- `key` (string)
- `index` (integer)
- **Returns:** the element; `0` if the key, list or index does not exist.

`st_lset(key, index, value)`

Set an element of a stored list, growing it if needed.

- **Parameters:**
- `key` (string)
- `index` (integer) - below 4194304. Gaps created by growing are filled with `0`.
- `value` (any)
- **Returns:** nothing.

```
bind start = do {
  st_lpush("queue", "a")
  st_lpush("queue", "b")
  st_lset("queue", 3, "d")
  print(st_llen("queue"))
  print(st_lget("queue", 1))
  print(st_lget("queue", 2))
}
```

```
OUTPUT
4
b
0
```

`st_ldel(key, index)`

Remove one element from a stored list.

- **Parameters:**
- `key` (string)
- `index` (integer)
- **Returns:** nothing. Later elements shift down; an out-of-range index does nothing.

`st_lclear(key)`

Replace a stored value with an empty list.

- **Parameters:**

- `key` (string)
- **Returns:** nothing.

Pixel drawing: the px_ primitives

The `px_*` builtins draw straight into the window's software framebuffer: fast, alpha-blended and clip-aware, with colours as packed `0xRRGGBB` numbers (the integer `255 * 65536 + 128 * 256 + 0` is orange). They were written for the DAW's 2D interface and need an open window; without one they draw nothing. All coordinates are pixels with the origin at the top left.

Every `px_*` drawing call honours the clip rectangle set by `px_clip`. The optional alpha argument is a number from 0 to 1 (default 1).

Builtin	Level	Summary
`px_clip(x, y, w, h)`	A	Restrict later `px_*` drawing to a rectangle.
`px_unclip()`	A	Remove the clip rectangle.
`px_rect(x, y, w, h, colour, alpha)`	A	Fill a rectangle.
`px_outline(x, y, w, h, colour, alpha)`	A	Draw a one-pixel rectangle outline.
`px_hline(x, y, length, colour, alpha)`	A	Draw a one-pixel horizontal line.
`px_vline(x, y, length, colour, alpha)`	A	Draw a one-pixel vertical line.
`px_rrect(x, y, w, h, radius, colour, alpha)`	A	Fill an anti-aliased rounded rectangle.
`px_rrect_outline(x, y, w, h, radius, colour, alpha)`	A	Draw an anti-aliased rounded-rectangle outline.
`px_disc(cx, cy, radius, colour, alpha)`	A	Fill an anti-aliased circle.
`px_ring(cx, cy, radius, colour, alpha)`	A	Draw an anti-aliased one-pixel circle outline.
`px_line(x0, y0, x1, y1, colour, alpha)`	A	Draw a one-pixel anti-aliased line.
`px_hatch(x, y, w, h, colour, angle, spacing, alpha)`	A	Fill a box with parallel one-pixel lines.
`px_bars(x0, mid, step, scale, peaks, colour, alpha)`	A	Draw a waveform as mirrored vertical bars.
`px_notes(notes, count, x, y, w, h, beat_width, lowest_key, key_span, loop_beats, view_x0, view_x1, colour_a, colour_b)`	A	Draw a piano-roll thumbnail of a clip.
`px_text(font, x, y, size, colour, text, crisp)`	A	Draw anti-aliased text.

Builtin	Level	Summary
`px_fit(font, size, max_width, text)`	A	Shorten text with ".." so it fits a width.
`px_color(colour)`	A	Set the pen colour used by text and `draw_poly`.
`px_theme(black)`	A	Switch the interface chrome greys to the near-black scheme.
`px_flat()`	A	Prepare the window for flat 2D drawing.
`px_snap_save()`	A	Remember the current framebuffer.
`px_snap_load()`	A	Restore the framebuffer saved by `px_snap_save`.

CHAPTER 25

Physics, Game Helpers and the Body Toolkit

This chapter covers the builtins that simulate things: soft and rigid bodies, a liquid grid, hyperbolic space, the 3D game helpers (textured meshes, collision, particles), the dialog box system and the acoustic body toolkit. They are backed by the crates `ling-physics`, `ling-game` and `ling-body`; the crates themselves are described in the crates chapter.

Conventions:

- **Handles** are integers counting from 0 in creation order. Calls with an unknown handle do nothing and return `0`, an empty list or unit; they never raise an error.
- **Defaults.** An omitted or non-numeric argument takes the default listed in the entry.
- **Headless use.** The simulation builtins (`soft_*`, `rb_*`, `liquid_step`, `body_*`, `coll_*`, `pfx_emit`, `pfx_step`, `dialog_*` state) run without a window. The drawing builtins (`liquid_draw`, `tm_draw`, `pfx_draw`, `dialog_draw`, `body_draw`, ...) draw into the open window and need one.
- **Names.** The `soft_*`, `rb_*` and `liquid_*` builtins also have translated names; each entry lists the Chinese (zh), Japanese (ja), Korean (ko) and Thai (th) ones. The `tm_*`, `coll_*`, `pfx_*` and `body_*` families have none.

Which way is up

Two coordinate conventions meet in this chapter, and mixing them up is the most common cause of "my body falls through the floor".

- The renderer is **y-down**: positive y is toward the bottom of the screen. The soft-body builtins follow it. A floor is at the *largest* y a body may reach, and gravity is a positive number.
- The rigid-body world (`rb_*`) is **y-up** inside the engine: its default gravity is `(0, -9.81, 0)` and `rb_floor` catches a body that is *above* the floor level and *moving toward negative y*.

If you flip `rb_gravity` to a positive y to match the renderer, `rb_floor` can no longer detect the floor, because it only reacts to a falling body (negative y velocity). Keep rigid bodies y-up and subtract y when you draw them, or leave `rb_floor` out and handle the floor yourself with

`rb_pos` and `rb_impulse`.

Soft bodies

A soft body is a spring-mass sphere (86 nodes) that squashes and stretches on impact. All soft bodies are independent of each other and of the rigid-body world, and they do not collide with each other.

Builtin	Level	Summary
`soft_ball(x, y, z, radius)`	B	Create a deformable ball.
`soft_step(id, dt, gravity)`	B	Advance a soft body by one time step.
`soft_bounce(id, floor_y, restitution)`	B	Collide a soft body with a horizontal floor.
`soft_contain(id, min_x, min_y, min_z, max_x, max_y, max_z, restitution)`	I	Keep a soft body inside a box.
`soft_kick(id, dx, dy, dz, strength)`	I	Give a soft body a push.
`soft_spin(id, ax, ay, az, rate)`	I	Add angular velocity about an axis through the centroid.
`soft_deform(id)`	I	How squashed the body is.
`soft_angular_speed(id)`	A	Tumbling speed.
`soft_centroid(id)`	B	Centre of mass.
`soft_nodes(id)`	B	Positions of every node.

`soft_ball(x, y, z, radius)`

Create a deformable ball.

- **Aliases:** zh `软球` · ja `ソフトボール` · ko `소프트볼` · th `ลูกบอลนุ่ม`
- **Parameters:**
- `x, y, z` (numbers, default 0) - centre.
- `radius` (number, default 1.0)
- **Returns:** soft-body id (integer from 0).

```
bind start = do {
  bind ball = soft_ball(0.0, 0.0 - 3.0, 0.0, 1.0)
  print(ball)
  print(len(soft_nodes(ball)) / 3)
}
```

```
OUTPUT
0
86
```

`soft_step(id, dt, gravity)`

Advance a soft body by one time step.

- **Aliases:** zh `软体步进` · ja `ソフト更新` · ko `소프트스텝` · th `ก้าวนุ่ม`
- **Parameters:**
- `id` (integer)
- `dt` (number, default 0.016) - seconds.
- `gravity` (number, default 15.0) - acceleration along +y. Positive pulls toward the floor of the screen.
- **Returns:** nothing.
- **Notes:** Four internal sub-steps run per call. Call it once per frame.

```
bind start = do {
  bind ball = soft_ball(0.0, 0.0 - 3.0, 0.0, 1.0)
  bind frame = 0
  while frame < 300 {
    soft_step(ball, 0.016, 9.8)
    soft_bounce(ball, 2.0, 0.5)
    bind frame = frame + 1
  }
  bind c = soft_centroid(ball)
  print("rests at y = " + str(round(c[1] * 100) / 100))
}
```

```
OUTPUT
rests at y = 1.09
```

`soft_bounce(id, floor_y, restitution)`

Collide a soft body with a horizontal floor.

- **Aliases:** zh `软体落地` · ja `ソフト着地` · ko `소프트바운스` · th `เด้งนุ่ม`
- **Parameters:**
- `id` (integer)
- `floor_y` (number, default 0) - the floor plane. With the y-down convention the body lives at `y <= floor_y`.
- `restitution` (number, default 0.5) - bounciness, limited to 0..0.99.
- **Returns:** nothing.
- **Notes:** A slow contact settles the node on the plane instead of reflecting it, so a resting ball does not creep upward.

`soft_contain(id, min_x, min_y, min_z, max_x, max_y, max_z, restitution)`

Keep a soft body inside a box.

- **Aliases:** zh `软体边界` · ja `ソフト箱` · ko `소프트경계` · th `กล่องนุ่ม`
- **Parameters:**
- `id` (integer)
- `min_x, min_y, min_z` (numbers, default -5)
- `max_x, max_y, max_z` (numbers, default 5)
- `restitution` (number, default 0.6)
- **Returns:** nothing.

```
GUI
bind start = do {
  bind ball = soft_ball(0.0, 0.0, 0.0, 1.0)
  soft_step(ball, 0.016, 9.0)
  soft_contain(ball, -4.0, -4.0, -4.0, 4.0, 4.0, 4.0, 0.55)
}
```

`soft_kick(id, dx, dy, dz, strength)`

Give a soft body a push.

- **Aliases:** zh `软体踢` · ja `ソフト衝撃` · ko `소프트킥` · th `เตะนุ่ม`
- **Parameters:**
- `id` (integer)
- `dx, dy, dz` (numbers, default 0) - direction; it is normalised, and a zero vector does nothing.
- `strength` (number, default 0.1) - speed gained along the direction.
- **Returns:** nothing.

`soft_spin(id, ax, ay, az, rate)`

Add angular velocity about an axis through the centroid.

- **Aliases:** zh `软体自旋` · ja `ソフト回転` · ko `소프트회전` · th `หมุนนุ่ม`
- **Parameters:**
- `id` (integer)
- `ax, ay, az` (numbers, default 0) - axis.
- `rate` (number, default 0.1) - radians per step. To roll without slipping use surface speed divided by radius.
- **Returns:** nothing.

`soft_deform(id)`

How squashed the body is.

- **Aliases:** zh `形变量` · ja `変形量` · ko `변형량` · th `ความบิดเบี้ยว`
- **Parameters:**
- `id` (integer)
- **Returns:** `0` for a perfect sphere, rising toward `1` when squished.

`soft_centroid(id)`

Centre of mass.

- **Aliases:** zh `软体质心` · ja `ソフト重心` · ko `소프트중심` · th `จุดศูนย์กลางนุ่ม`
- **Parameters:**

- `id` (integer)
- **Returns:** list `[x, y, z]`; `[0, 0, 0]` for an unknown id.

`soft_nodes(id)`

Positions of every node.

- **Aliases:** zh `软体节点` · ja `ソフト節点` · ko `소프트노드` · th `จุดนุ่ม`
- **Parameters:**
- `id` (integer)
- **Returns:** flat list `[x, y, z, x, y, z, ...]` of 86 nodes. Draw it to show the deformed surface.

Rigid bodies

One shared world of axis-aligned boxes (half-size 0.5 on every axis) with linear and angular dynamics. Bodies collide with each other with a simple O(n squared) test. See the note on y-up earlier in this chapter.

Builtin	Level	Summary
`rb_add(x, y, z, mass)`	B	Add a rigid body to the world.
`rb_gravity(gx, gy, gz)`	I	Set the world gravity.
`rb_step(dt)`	B	Advance the whole world.
`rb_floor(index, floor_y, restitution, friction)`	B	Bounce a body off a horizontal floor.
`rb_impulse(index, ix, iy, iz)`	I	Apply a linear impulse.
`rb_torque(index, tx, ty, tz)`	I	Apply a torque for the next step.
`rb_spin(index, wx, wy, wz)`	I	Add to the angular velocity.
`rb_pos(index)`	B	Position of a body.
`rb_rot(index)`	I	Orientation of a body.

`rb_add(x, y, z, mass)`

Add a rigid body to the world.

- **Aliases:** zh `刚体添加` · ja `剛体追加` · ko `강체추가` · th `เพิ่มวัตถุแข็ง`
- **Parameters:**

- `x, y, z` (numbers, default 0) - start position.
- `mass` (number, default 1.0)
- **Returns:** body index.
- **Notes:** New bodies have a restitution of 0.6 and a unit-cube extent.

`rb_gravity(gx, gy, gz)`

Set the world gravity.

- **Aliases:** zh `刚体重力` · ja `剛体重力` · ko `강체중력` · th `แรงโน้มถ่วงแข็ง`
- **Parameters:**
- `gx` (number, default 0)
- `gy` (number, default 9.81) - mind the sign: the engine default before any call is `-9.81`, but an omitted `gy` here becomes `+9.81`.
- `gz` (number, default 0)
- **Returns:** nothing.
- **Notes:** Applies to the whole world.

`rb_step(dt)`

Advance the whole world.

- **Aliases:** zh `刚体步进` · ja `剛体更新` · ko `강체스텝` · th `ก้าวแข็ง`
- **Parameters:**
- `dt` (number, default 0.016) - seconds.
- **Returns:** nothing. Integrates every body (including rotation) and resolves body-body collisions.

`rb_floor(index, floor_y, restitution, friction)`

Bounce a body off a horizontal floor.

- **Aliases:** zh `刚体落地` · ja `剛体着地` · ko `강체바닥` · th `พื้นแข็ง`
- **Parameters:**
- `index` (integer)
- `floor_y` (number, default 0)
- `restitution` (number, default 0.6)
- `friction` (number, default 0.6) - converts sideways motion into roll.

- **Returns:** nothing.
- **Notes:** Call it every frame for every body that should be able to hit the floor. It acts only when the body's lower face is below `floor_y` and it is moving toward negative y; see the y-up note.

```
bind start = do {
  bind box = rb_add(0.0, 5.0, 0.0, 1.0)
  rb_impulse(box, 2.0, 0.0, 0.0)
  bind frame = 0
  while frame < 240 {
    rb_step(0.016)
    rb_floor(box, 0.0, 0.5, 0.6)
    bind frame = frame + 1
  }
  bind p = rb_pos(box)
  print("x = " + str(round(p[0] * 100) / 100))
  print("y = " + str(round(p[1] * 100) / 100))
}
```

```
OUTPUT
x = 4.5
y = 0.5
```

`rb_impulse(index, ix, iy, iz)`

Apply a linear impulse.

- **Aliases:** zh `刚体冲量` · ja `剛体インパルス` · ko `강체충격` · th `แรงดลแข็ง`
- **Parameters:**
- `index` (integer)
- `ix, iy, iz` (numbers, default 0) - velocity change times mass.
- **Returns:** nothing.

`rb_torque(index, tx, ty, tz)`

Apply a torque for the next step.

- **Aliases:** zh `扭矩` · ja `トルク` · ko `토크` · th `แรงบิด`

- **Parameters:**
- `index` (integer)
- `tx, ty, tz` (numbers, default 0) - world axes.
- **Returns:** nothing.

`rb_spin(index, wx, wy, wz)`

Add to the angular velocity.

- **Aliases:** zh `自旋` · ja `スピン` · ko `스핀` · th `หมุน`
- **Parameters:**
- `index` (integer)
- `wx, wy, wz` (numbers, default 0) - radians per second about the world axes.
- **Returns:** nothing.

`rb_pos(index)`

Position of a body.

- **Aliases:** zh `刚体位置` · ja `剛体位置` · ko `강체위치` · th `ตำแหน่งแข็ง`
- **Parameters:**
- `index` (integer)
- **Returns:** list `[x, y, z]`.

`rb_rot(index)`

Orientation of a body.

- **Aliases:** zh `刚体旋转` · ja `剛体回転` · ko `강체회전` · th `การหมุนแข็ง`
- **Parameters:**
- `index` (integer)
- **Returns:** quaternion list `[x, y, z, w]`; `[0, 0, 0, 1]` is no rotation. Rotate your mesh by it to draw a tumbling body.

Liquid simulation

A fast 2D grid fluid with two immiscible liquids, water and oil, advected by one incompressible velocity field. The liquids blend only at their interface, so the physics decides where the two colours mix. The grid wraps around at its edges, which makes it seamless when mapped onto a sphere or cylinder.

Builtin	Level	Summary
`liquid_new(width, height)`	B	Create a fluid grid.
`liquid_splat(id, x, y, kind, amount, radius)`	B	Add fluid in a disc.
`liquid_gravity(id, gx, gy)`	I	Set the gravity direction of a grid.
`liquid_step(id, dt)`	B	Advance one grid.
`liquid_step_all(dt)`	A	Advance every grid in parallel.
`liquid_mix(id)`	I	How mixed the two liquids are.
`liquid_set_colors(id, wr, wg, wb, or, og, ob)`	I	Set the water and oil colours.
`liquid_rainbow(id, on)`	A	Colour the fluid as a flowing seven-colour marble.
`liquid_draw(id, x, y, scale)`	I	Blit the colour field to the window.
`liquid_draw_surface(id, kind, cx, cy, cz, radius, height)`	I	Map the fluid onto a 3D surface.

`liquid_new(width, height)`

Create a fluid grid.

- **Aliases:** zh `新建液体` · ja `液体新規` · ko `액체생성` · th `สร้างของเหลว`
- **Parameters:**
- `width` (integer, default 64) - cells; at least 2.
- `height` (integer, default 64) - cells; at least 2.
- **Returns:** grid id.

`liquid_splat(id, x, y, kind, amount, radius)`

Add fluid in a disc.

- **Aliases:** zh `液体注入` · ja `液体追加` · ko `액체분사` · th `หยดของเหลว`
- **Parameters:**
- `id` (integer)

- `x, y` (numbers, default 0) - centre in cells.
- `kind` (integer, default 0) - `0` water, `1` oil.
- `amount` (number, default 1.0)
- `radius` (number, default 4.0) - cells.
- **Returns:** nothing.

`liquid_gravity(id, gx, gy)`

Set the gravity direction of a grid.

- **Aliases:** zh `液体重力` · ja `液体重力ベクトル` · ko `액체중력` · th `แรงโน้มถ่วงเหลว`
- **Parameters:**
- `id` (integer)
- `gx` (number, default 0)
- `gy` (number, default 60)
- **Returns:** nothing.
- **Notes:** Rotate this vector over time to make the liquids slosh.

`liquid_step(id, dt)`

Advance one grid.

- **Aliases:** zh `液体步进` · ja `液体更新` · ko `액체스텝` · th `ก้าวของเหลว`
- **Parameters:**
- `id` (integer)
- `dt` (number, default 0.016) - seconds.
- **Returns:** nothing.

`liquid_mix(id)`

How mixed the two liquids are.

- **Aliases:** zh `液体混合` · ja `液体混合度` · ko `액체혼합` · th `การผสมของเหลว`
- **Parameters:**
- `id` (integer)
- **Returns:** `0` for fully separated up to `1` for fully intermixed.

```
bind start = do {
  bind pool = liquid_new(16, 16)
  liquid_splat(pool, 8.0, 4.0, 1, 1.0, 3.0)
  liquid_splat(pool, 8.0, 12.0, 0, 1.0, 3.0)
  print(liquid_mix(pool))
  liquid_gravity(pool, 0.0, 60.0)
  bind frame = 0
  while frame < 50 {
    liquid_step(pool, 0.016)
    bind frame = frame + 1
  }
  print(round(liquid_mix(pool) * 1000) / 1000)
}
```

```
OUTPUT
0
0.138
```

`liquid_set_colors(id, wr, wg, wb, or, og, ob)`

Set the water and oil colours.

- **Aliases:** zh `液体颜色` · ja `液体配色` · ko `액체색상` · th `สีของเหลว`
- **Parameters:**
- `id` (integer)
- `wr, wg, wb` (numbers 0..255, default 40, 110, 235) - water.
- `or, og, ob` (numbers 0..255, default 240, 175, 45) - oil.
- **Returns:** nothing.

`liquid_draw(id, x, y, scale)`

Blit the colour field to the window.

- **Aliases:** zh `绘制液体` · ja `液体描画` · ko `액체그리기` · th `วาดของเหลว`
- **Parameters:**
- `id` (integer)
- `x, y` (integers, default 0) - top-left pixel.

- `scale` (integer, default 4) - pixels per cell, at least 1.
- **Returns:** nothing.
- **Notes:** Writes pixels directly into the framebuffer, ignoring the 3D depth queue.

```
GUI
bind start = do {
  open_window(480, 320)
  bind pool = liquid_new(84, 84)
  liquid_splat(pool, 42.0, 25.0, 1, 1.0, 3.0)
  liquid_splat(pool, 42.0, 58.0, 0, 1.0, 3.0)
  liquid_step(pool, 0.016)
  liquid_draw(pool, 20, 20, 3)
  present()
}
```

`liquid_draw_surface(id, kind, cx, cy, cz, radius, height)`

Map the fluid onto a 3D surface.

- **Aliases:** zh `液体贴面` · ja `液体曲面` · ko `액체곡면` · th `ของเหลวบนพื้นผิว`
- **Parameters:**
- `id` (integer)
- `kind` (integer, default 1) - `0` plane, `1` sphere, `2` cylinder, `3` cone, `4` dome.
- `cx, cy, cz` (numbers, default 0) - centre of the surface.
- `radius` (number, default 2.0)
- `height` (number, default 3.0)
- **Returns:** nothing.

```
GUI
bind start = do {
  open_window(480, 320)
  bind pool = liquid_new(64, 64)
  liquid_splat(pool, 32.0, 16.0, 1, 1.0, 4.0)
  liquid_step(pool, 0.016)
  set_camera(cos(0.3), sin(0.3), cos(0.4), sin(0.4))
  liquid_draw_surface(pool, 1, 0.0, 0.0, 0.0, 2.2, 3.2)
  present()
}
```

Hyperbolic space

Ling can bend the 3D view into hyperbolic space. These builtins are view effects: they change how the scene is projected, not the coordinates the scene is built in. The underlying maths (Poincare ball, Mobius addition, exponential and logarithmic maps) is in the `ling_physics::hyperbolic` module, covered in the crates chapter.

Builtin	Level	Summary
`set_camera_hyper(radius, saddle, px, mid_y, pz)`	A	Warp the view into hyperbolic space around a pivot.
`tm_screen_lens(k)`	A	Poincare-disk lens over the finished frame.
`vtex_hyperbolic_uv(cx, cy, cz, ux, uy, uz, vx, vy, vz, max_r, circles, rays, fr, hue)`	A	Draw a Poincare-disk pattern of circles and rays on a plane.

`tm_tex_load(path, size)`

Load a texture.

- **Parameters:**
- `path` (string) - SVG, SVGZ or PNG; resolved relative to the script folder when not found as given.
- `size` (number, default 256) - side length in pixels, rounded to a power of two.
- **Returns:** texture id (integer from 0), or `-1` on failure (the reason goes to standard error).

```
GUI
bind start = do {
  bind rock = tm_tex_load("rock.png", 256)
  print(rock)
}
```

`tm_load(path)`

Load an `.ltm` mesh.

- **Parameters:**
- `path` (string)
- **Returns:** mesh id (integer from 0), or `-1` on failure.

`tm_groups(mesh)`

Number of groups in a mesh.

- **Parameters:**
- `mesh` (integer)
- **Returns:** integer; `0` for an unknown mesh.

`tm_group(mesh, name)`

Find a group by name.

- **Parameters:**
- `mesh` (integer)
- `name` (string)
- **Returns:** index of the group with exactly that name, otherwise of the first group whose name contains it, otherwise `-1`.

`tm_group_name(mesh, group)`

Name of a group.

- **Parameters:**
- `mesh` (integer)

- `group` (integer)
- **Returns:** string; empty for an invalid index.

`tm_group_tris(mesh, group)`

Triangle count of a group.

- **Parameters:**
- `mesh` (integer)
- `group` (integer)
- **Returns:** integer.

`tm_bounds(mesh, group)`

Bounding box.

- **Parameters:**
- `mesh` (integer)
- `group` (integer, default -1) - a group, or -1 for the whole mesh.
- **Returns:** list `[min_x, min_y, min_z, max_x, max_y, max_z]`. For an unknown mesh the minimum values are `+infinity` and the maximum values `-infinity`.

`tm_draw(mesh, group, tex, x, y, z, scale, rot, tint, alpha, uoff, voff, uscale, vscale, shade, blend, cull, ambient)`

Queue a mesh group for drawing.

- **Parameters:**
- `mesh` (integer) - mesh id.
- `group` (integer, default -1) - a group, or -1 for all groups.
- `tex` (integer, default -1) - texture id; -1 is the glow dot.
- `x, y, z` (numbers, default 0) - world position of the mesh origin.
- `scale` (number, default 1)
- `rot` (number or list, default 0) - a yaw angle in radians, or a row-major 3x3 rotation matrix as a list of nine numbers.
- `tint` (number, default 0xFFFFFF) - packed colour that multiplies the texture.
- `alpha` (number, default 1) - 0 to 1.
- `uoff, voff` (numbers, default 0) - texture scroll.

- `uscale, vscale` (numbers, default 1) - texture repeat.
- `shade` (number 0..1, default 0) - darkens faces turned away from a fixed light.
- `blend` (integer, default 0) - 0 normal, 1 additive.
- `cull` (integer, default 0) - 0 draws both sides, 1 or 2 culls one winding.
- `ambient` (number, default -1) - 0 or more switches on the scene lights with this ambient level; negative means unlit.
- **Returns:** number of triangles queued.
- **Notes:** Triangles are perspective-correct, near-clipped, fogged when fog is set, and opaque draws write depth. A see-through circle set with `tm_hole` fades surfaces in front of its point, and `tm_ripple_use(1)` adds floor shock-waves to the draws that follow.

```
GUI
bind start = do {
  open_window(640, 360)
  bind rock = tm_tex_load("rock.png", 256)
  bind level = tm_load("level.ltm")
  tm_draw(level, -1, rock, 0.0, 0.0, 10.0, 1.0, 0.0, 16777215, 1.0, 0.0, 0.0, 1.0, 1.0,
0.3, 0, 1, -1.0)
  flush_3d()
  present()
}
```

`tm_sprite(tex, cx, cy, w, h, rot, tint, alpha, blend)`

Draw a rotated textured quad on the 2D framebuffer.

- **Parameters:**
- `tex` (integer, default -1)
- `cx, cy` (numbers, default 0) - centre in pixels.
- `w, h` (numbers, default 64) - size in pixels.
- `rot` (number, default 0) - radians.
- `tint` (number, default 0xFFFFFF)
- `alpha` (number, default 1)
- `blend` (integer, default 0) - 1 is additive.
- **Returns:** nothing.
- **Notes:** Draws immediately onto the framebuffer, so call it after `flush_3d`. Use it for title cards and HUD art.

`tm_billboard(tex, x, y, z, size, rot, tint, alpha, blend)`

Draw a camera-facing textured quad at a world point.

- **Parameters:**
- `tex` (integer, default -1)
- `x, y, z` (numbers, default 0)
- `size` (number, default 1)
- `rot` (number, default 0) - radians.
- `tint` (number, default 0xFFFFFF)
- `alpha` (number, default 1)
- `blend` (integer, default 0)
- **Returns:** nothing.
- **Notes:** When `tm_lens_guard` is active, the quad fades out as the eye comes within its own size and is skipped on top of the lens.

`tm_wire(kind, x, y, z, size, rx, ry, rz, colour, detail, thick)`

Draw a wireframe shape as depth-sorted 3D lines.

- **Parameters:**
- `kind` (integer, default 0) - 0 cube, 1 tetrahedron, 2 octahedron, 3 icosahedron, 4 torus, 5 sphere (`detail` rings), 6 ring (`detail` sides, in the XY plane), 7 star prism, 8 grid (`detail` by `detail`, in the XZ plane).
- `x, y, z` (numbers, default 0) - centre.
- `size` (number, default 1) - radius.
- `rx, ry, rz` (numbers, default 0) - rotation in radians, applied as Z, then Y, then X.
- `colour` (number, default 0xFFFFFF)
- `detail` (integer, default 12) - clamped to 3..64.
- `thick` (number, default 0) - above 0.5 doubles every stroke one pixel over, for a bolder line under bloom.
- **Returns:** nothing.
- **Notes:** Drawn in the current blend mode; call `set_blend(1)` first for additive neon.

```
GUI
bind start = do {
  open_window(640, 360)
  set_blend(1)
  tm_wire(3, 0.0, 0.0, 8.0, 2.0, 0.3, 0.6, 0.0, 65535, 12, 1)
  flush_3d()
  present()
}
```

`tm_solid(kind, x, y, z, size, rx, ry, rz, colour, alpha, blend, stretch)`

Draw a faceted, flat-shaded low-poly solid.

- **Parameters:**
- `kind` (integer, default 3) - 0 cube, 1 tetrahedron, 2 octahedron, 3 icosahedron, 9 shard (an octahedron stretched along its local y).
- `x, y, z` (numbers, default 0)
- `size` (number, default 1)
- `rx, ry, rz` (numbers, default 0) - radians.
- `colour` (number, default 0xFFFFFF)
- `alpha` (number, default 1) - below 1 is translucent.
- `blend` (integer, default 0) - 1 adds, for glowing glass.
- `stretch` (number, default 1; 2.4 for the shard) - stretch along local y.
- **Returns:** nothing.
- **Notes:** Every face gets its own tone from a fixed key light: the crystal and shard look.

`tm_fov(degrees, viewport_width)`

Set the horizontal field of view.

- **Parameters:**
- `degrees` (number, default 0) - between 1 and 179; any other value restores the default (focal length equal to the window height).
- `viewport_width` (number, default: window width) - pixel width the angle spans.
- **Returns:** nothing.

`tm_collide(ax, ay, ar, bx, by, br)`

Find every overlapping pair of circles between two sets.

- **Parameters:**
- `ax, ay` (lists of numbers) - x and y of each circle in set A.
- `ar` (number or list) - radius for all of A, or one radius per circle.
- `bx, by` (lists of numbers) - set B.
- `br` (number or list) - radius of B.
- **Returns:** flat list `[i0, j0, i1, j1, ...]` of index pairs `(i in A, j in B)` whose circles overlap. Native, so a shooter can test every bullet against every rock in one call.

```
bind start = do {
  bind hits = tm_collide([0.0, 5.0], [0.0, 5.0], 1.0, [0.5, 9.0], [0.5, 9.0], 1.0)
  print(hits)
}
```

```
OUTPUT
[0, 0]
```

`coll_build(mesh, group, x, y, z, scale, rot, cell)`

Build a collision shape from a mesh.

- **Parameters:**
- `mesh` (integer)
- `group` (integer, default -1) - or -1 for all groups.
- `x, y, z` (numbers, default 0) - placement offset.
- `scale` (number, default 1)
- `rot` (number or list, default 0) - yaw in radians or a row-major 3x3 matrix, as for `tm_draw`.
- `cell` (number, default 8) - size of the lookup grid cells, at least 0.5.
- **Returns:** collision id (integer), or `-1` for an unknown mesh. Degenerate triangles are dropped.
- **Notes:** The shape is static; build it once with the final transform.

`coll_sphere(id, x, y, z, radius)`

Push a sphere out of a collision shape.

- **Parameters:**
- `id` (integer)
- `x, y, z` (numbers, default 0) - sphere centre.
- `radius` (number, default 1.0)
- **Returns:** list `[x, y, z, nx, ny, nz, contacts, deepest]` : the corrected centre, the summed contact normal (normalised, zeros if there was no contact), the number of triangle contacts and the deepest penetration. Up to four relaxation passes run per call. For an invalid id the centre is returned unchanged with zeros after it.

`coll_ray(id, ox, oy, oz, dx, dy, dz, max_t)`

Cast a ray against a collision shape.

- **Parameters:**
- `id` (integer)
- `ox, oy, oz` (numbers) - origin.
- `dx, dy, dz` (numbers, default 0, 0, 1) - direction; it is normalised.
- `max_t` (number, default 1000) - maximum distance.
- **Returns:** list `[t, nx, ny, nz]` with the hit distance and the triangle normal, or `[-1, 0, 0, 0]` for no hit.

```
bind start = do {
  file_write_hex("floor.ltm",
"4c544d310100000005666c6f6f7204000000020000000000a0c0000000000000a0c0000000000000000000000000
  bind mesh = tm_load("floor.ltm")
  bind floor = coll_build(mesh, -1, 0.0, 0.0, 0.0, 1.0, 0.0, 8.0)
  print("triangles: " + str(coll_tris(floor)))
  print(coll_ray(floor, 0.0, -3.0, 0.0, 0.0, 1.0, 0.0, 10.0))
  print(coll_ray(floor, 0.0, -3.0, 0.0, 0.0, 1.0, 0.0, 1.0))
  bind hit = coll_sphere(floor, 0.0, 0.2, 0.0, 0.5)
  print(round(hit[1] * 100) / 100)
  print(hit[6])
}
```

```
OUTPUT
triangles: 2
[3, 0, -1, 0]
[-1, 0, 0, 0]
0.5
1
```

`pfx_emit(x, y, z, vx, vy, vz, life, size0, size1, col0, col1, kind, tex, gravity, alpha)`

Emit one particle.

- **Parameters:**
- `x, y, z` (numbers, default 0) - start position.
- `vx, vy, vz` (numbers, default 0) - initial velocity.
- `life` (number, default 1.0) - seconds.
- `size0, size1` (numbers, default 0.2 and 0) - size at birth and at death.
- `col0, col1` (numbers, default 0xFFFFFF) - packed colours at birth and death; `col1` defaults to `col0`.
- `kind` (integer, default 0) - 0 glow dot (additive), 1 streak (additive), 2 sprite (uses `tex`, normal blend).
- `tex` (integer, default 0) - texture id for kind 2.
- `gravity` (number, default 1.0) - multiplier of the gravity passed to `pfx_step`.
- `alpha` (number, default 1.0)
- **Returns:** nothing.

`pfx_burst(x, y, z, count, speed, life, size0, size1, col0, col1, kind, gravity, vx, vy, vz)`

Emit a burst of particles in random directions, like a firework shell.

- **Parameters:**
- `x, y, z` (numbers, default 0) - origin.
- `count` (integer, default 16) - clamped to 0..2000.
- `speed` (number, default 5.0) - each particle gets 0.55 to 1.0 times this.
- `life` (number, default 1.0) - each particle lives 0.6 to 1.0 times this.
- `size0, size1` (numbers, default 0.2 and 0)

- `col0, col1` (numbers, default 0xFFFFFF)
- `kind` (integer, default 0) - as for `pfx_emit`.
- `gravity` (number, default 1.0)
- `vx, vy, vz` (numbers, default 0) - velocity inherited by every particle.
- **Returns:** nothing.
- **Notes:** Directions and speeds come from the particle system's own fixed-seed generator, not from `rand()`: a program that makes the same calls produces the same bursts every run, and `rand_seed` does not change them.

`pfx_step(dt, gravity, drag, floor_y, bounce)`

Advance every particle.

- **Parameters:**
- `dt` (number, default 1/60) - seconds.
- `gravity` (number, default 9.8) - acceleration along +y (down), scaled per particle by its `gravity` multiplier.
- `drag` (number, default 0) - velocity loss per second.
- `floor_y` (number, default 1e9) - particles below this y bounce.
- `bounce` (number, default 0.4) - fraction of speed kept on a bounce.
- **Returns:** nothing. Expired particles are removed.

```
bind start = do {
  pfx_burst(0.0, 0.0, 0.0, 20, 5.0, 1.0, 0.2, 0.0, 16777215, 255, 0, 1.0, 0.0, 0.0,
0.0)
  print(pfx_count())
  pfx_step(0.5)
  print(pfx_count())
  pfx_step(1.0)
  print(pfx_count())
}
```

```
OUTPUT
20
20
0
```

`pfx_draw()`

Draw all live particles into the depth queue.

- **Returns:** nothing.

```
GUI
bind start = do {
  open_window(640, 360)
  pfx_burst(0.0, 0.0, 8.0, 40, 4.0, 1.5, 0.3, 0.0, 16776960, 16711680, 0, 1.0, 0.0,
0.0, 0.0)
  pfx_step(0.016)
  pfx_draw()
  flush_3d()
  present()
}
```

`pfx_clear()`

Remove all particles.

- **Returns:** nothing.

`pfx_count()`

Number of live particles.

- **Returns:** integer.

`dialog_show(text, chars_per_second)`

Start a dialog.

- **Aliases:** zh `对话显示` · ja `会話表示` · ko `대화표시` · th `แสดงบทสนทนา`
- **Parameters:**
- `text` (string) - text with the markup above.
- `chars_per_second` (number, default 32) - typing speed, at least 1.
- **Returns:** nothing.
- **Notes:** Replaces any dialog already open.

`dialog_step(dt)`

Advance the typewriter and the auto-advance timer.

- **Aliases:** zh `对话步进` · ja `会話更新` · ko `대화스텝` · th `ก้าวบทสนทนา`
- **Parameters:**
- `dt` (number, default 0.016) - seconds.
- **Returns:** nothing. Call it every frame.

`dialog_advance()`

Respond to the player pressing the confirm key.

- **Aliases:** zh `对话推进` · ja `会話送り` · ko `대화진행` · th `เลื่อนบทสนทนา`
- **Returns:** nothing. If the page is still typing it is revealed at once; otherwise the dialog moves to the next page, or closes after the last page.

`dialog_active()`

Whether a dialog is open.

- **Aliases:** zh `对话激活` · ja `会話中` · ko `대화중` · th `บทสนทนาทำงาน`
- **Returns:** `true` or `false` (a real boolean).

`dialog_typing()`

Whether the current page is still being typed.

- **Aliases:** zh `对话打字` · ja `会話タイプ中` · ko `대화타이핑` · th `กำลังพิมพ์บทสนทนา`
- **Returns:** `true` or `false`.

`dialog_close()`

Close the dialog immediately.

- **Aliases:** zh `对话关闭` · ja `会話閉じる` · ko `대화닫기` · th `ปิดบทสนทนา`
- **Returns:** nothing.

`dialog_color(role, r, g, b)`

Set the colour of a text role.

- **Aliases:** zh `对话颜色` · ja `会話色` · ko `대화색` · th `สีบทสนทนา`
- **Parameters:**
- `role` (integer) - 0 plain text, 1 name, 2 place, 3 item; values above 3 are treated as 3.
- `r, g, b` (integers 0..255, default 255)
- **Returns:** nothing.

`dialog_draw(x, y, width, height, font)`

Draw the dialog box and its revealed text.

- **Aliases:** zh `对话绘制` · ja `会話描画` · ko `대화그리기` · th `วาดบทสนทนา`
- **Parameters:**
- `x, y` (numbers, default 40 and 0) - top-left of the box.
- `width, height` (numbers, default 720 and 150)
- `font` (integer, default -1) - a font handle from `font_load`; a negative value uses the runtime's fallback text drawing.
- **Returns:** nothing.

```
GUI
bind start = do {
  open_window(800, 300)
  dialog_show("{n}Ling{/} arrives at {p}the harbour{/}.||She finds a {i}silver
key{/}.", 30)
  dialog_step(0.5)
  dialog_draw(40, 120, 720, 150)
  present()
}
```

Stepping a dialog without a window

```
bind start = do {
  dialog_show("Hello {n}Ling{/}.||Second page.", 100)
  print("active: " + str(dialog_active()))
  print("typing: " + str(dialog_typing()))
  dialog_step(0.5)
  print("typing after 0.5 s: " + str(dialog_typing()))
  dialog_advance()
  print("page 2 typing: " + str(dialog_typing()))
  dialog_advance()
  dialog_advance()
  print("active at the end: " + str(dialog_active()))
}
```

```
OUTPUT
active: true
typing: true
typing after 0.5 s: false
page 2 typing: true
active at the end: false
```

Small utilities in the game helpers

These builtins live in the same module as the `tm_` family but have nothing to do with meshes. They are handy in games with save codes, networking or shared random worlds.

Builtin	Level	Summary
`str_to_hex(text)`	I	Encode text as lowercase hexadecimal.
`hex_to_str(hex)`	I	Decode hexadecimal to text.
`random_hex(bytes)`	I	Random bytes from the operating system, as hex.
`rand_seed(n)`	I	Reseed `rand()`.
`net_local_ip()`	A	This machine's LAN address.
`clipboard_set(text)`	A	Put text on the system clipboard.
`clipboard_get()`	A	Read the system clipboard.

`str_to_hex(text)`

Encode text as lowercase hexadecimal.

- **Parameters:**
- `text` (string)
- **Returns:** hex string, two characters per UTF-8 byte.
- **Notes:** The `p2p_*` payloads use this encoding.

```
GUI
bind start = do {
  print(str_to_hex("Ling"))
  print(hex_to_str("4c696e67"))
}
```

`hex_to_str(hex)`

Decode hexadecimal to text.

- **Parameters:**
- `hex` (string)
- **Returns:** the decoded string; invalid pairs are skipped and invalid UTF-8 is replaced.

`random_hex(bytes)`

Random bytes from the operating system, as hex.

- **Parameters:**
- `bytes` (integer, default 32) - clamped to 1..256.
- **Returns:** hex string of length `2 * bytes`.
- **Notes:** Use this for keys and seeds. `rand()` is not for secrets.

```
bind start = do {
  print(len(random_hex(8)))
}
```

```
OUTPUT
16
```

`rand_seed(n)`

Reseed `rand()`.

- **Parameters:**
- `n` (number, default 1)
- **Returns:** nothing.
- **Notes:** Two machines that call `rand_seed` with the same number and then make the same sequence of `rand()` calls build the same world.

`body_message()`

Message left by the last operation.

- **Returns:** string. After a success it describes the result (for example `"mesh updated"`); after a failure it holds the reason.

`body_param_get(name)`

Current value of a parameter.

- **Parameters:**
- `name` (string)
- **Returns:** number; `0` for an unknown name.

`body_param_set(name, value)`

Set a parameter.

- **Parameters:**
- `name` (string)
- `value` (number) - clamped to the parameter's range.
- **Returns:** `1` if the name exists, `0` if not.
- **Notes:** Changing a body or simulation parameter discards the current rest shape and simulation; changing a medium, microphone or membrane parameter discards the simulation.

```
bind start = do {
  print(body_param_set("height", 1.9))
  print(body_param_get("height"))
  print(body_param_set("height", 9))
  print(body_param_get("height"))
  print(body_param_set("nope", 1))
}
```

```
OUTPUT
1
1.9
1
2.4
0
```

`body_param_reset()`

Restore every parameter to its default and remove all sources.

- **Returns:** nothing.

Acoustic sources

A source is an excitation placed in space: a position, a waveform, a frequency, an amplitude and a start time and duration. Fields: `x`, `y`, `z` (metres), `wave` (0 Ricker wavelet, 1 sine burst, 2 impulse), `freq` (Hz, 1 to 40000), `amp`, `phase`, `start` (seconds after the start, at least 0) and `dur` (seconds).

Builtin	Level	Summary
`body_src_count()`	I	Number of sources.
`body_src_add(x, y, z, wave, freq)`	I	Add a source at a position.
`body_src_at_joint(joint, wave, freq)`	I	Add a source at a named joint of the scaffold.
`body_src_del(index)`	I	Remove a source.
`body_src_clear()`	I	Remove all sources.
`body_src_get(index, field)`	I	Read a source field.
`body_src_set(index, field, value)`	I	Change a source field.

Builtin	Level	Summary
`body_src_name(index)`	A	Name of a source.

`body_src_count()`

Number of sources.

- **Returns:** integer.

`body_src_add(x, y, z, wave, freq)`

Add a source at a position.

- **Parameters:**
- `x, y, z` (numbers, default 0, 1, 0) - metres; the default is one metre above the origin, near the chest.
- `wave` (integer, default 0) - 0 Ricker, 1 sine burst, 2 impulse.
- `freq` (number, default 220) - Hz.
- **Returns:** index of the new source.

`body_src_at_joint(joint, wave, freq)`

Add a source at a named joint of the scaffold.

- **Parameters:**
- `joint` (string) - a name from `body_joint_name`, such as `"hips"`.
- `wave` (integer, default 0)
- `freq` (number, default 220)
- **Returns:** index of the new source, or `-1` for an unknown joint. The source is named after the joint.

```
bind start = do {
  print(body_src_at_joint("hips", 0, 110))
  print(body_src_name(0))
  print(body_src_get(0, "freq"))
  print(body_src_at_joint("no_such_joint"))
}
```

```
OUTPUT
0
hips
110
-1
```

`body_src_del(index)`

Remove a source.

- **Parameters:**
- `index` (integer)
- **Returns:** nothing.

`body_src_clear()`

Remove all sources.

- **Returns:** nothing.

`body_src_get(index, field)`

Read a source field.

- **Parameters:**
- `index` (integer)
- `field` (string) - `"x"`, `"y"`, `"z"`, `"wave"`, `"freq"`, `"amp"`, `"phase"`, `"start"` or `"dur"`.
- **Returns:** number; `0` for an unknown index or field.

`body_src_set(index, field, value)`

Change a source field.

- **Parameters:**
- `index` (integer)
- `field` (string) - as for `body_src_get`.
- `value` (number)
- **Returns:** `1` on success, `0` for an unknown index or field.

- **Notes:** Any change discards the running simulation.

`body_preview()`

Voxelise the body and extract its rest shape without simulating.

- **Returns:** `1` on success, `0` on failure. After it, `body_mesh_counts`, `body_mesh_bounds` and `body_draw` show the undisturbed body.

`body_sim_begin()`

Start a simulation with the current parameters and sources.

- **Returns:** total number of steps, or `-1` on failure (see `body_message`).

`body_sim_step(budget_ms)`

Run the simulation for about a time budget.

- **Parameters:**
- `budget_ms` (number, default 30) - milliseconds of work to do in this call.
- **Returns:** list `[steps_done, steps_total]`; `[0, 0]` if no simulation was started.

`body_sim_active()`

Whether a simulation exists.

- **Returns:** `1` or `0`.

`body_sim_done()`

Whether the simulation has run all its steps.

- **Returns:** `1` or `0`.

`body_mesh_update()`

Extract the membrane's current surface as the working mesh.

- **Returns:** `1` on success, `0` if no simulation exists.

- **Notes:** Callable while the simulation is running to watch the skin evolve. Once the simulation is complete it also captures the microphone signals for `body_export_wav`.

`body_mesh_counts()`

Size of the working mesh.

- **Returns:** list `[vertices, triangles, watertight, recovered_from_wav]`, the last two as `1` or `0`.

`body_mesh_bounds()`

Bounding box of the body.

- **Returns:** list `[min_x, min_y, min_z, max_x, max_y, max_z]` in metres; zeros if there is no mesh.

`body_export_mesh(path)`

Write the working mesh to a file.

- **Parameters:**
- `path` (string) - `.obj`, `.stl` or `.glb`. A `.glb` includes a skeleton fitted to the body.
- **Returns:** `1` on success, `0` on failure (an unknown extension or no mesh). The message gives the triangle count.

`body_export_wav(path)`

Write the simulated sound and the body recipe to a WAV file.

- **Parameters:**
- `path` (string)
- **Returns:** `1` on success, `0` if the simulation has not run to completion. The file embeds the parameters and the mesh, so it can be loaded back with `body_import_wav`.
- **Notes:** The signal is tiled `wav_repeat` times with `wav_gap` seconds between repeats, at `wav_bits` bits and `wav_gain` gain.

`body_import_wav(path, analyze_only)`

Load a WAV as a body.

- **Parameters:**
- `path` (string)
- `analyze_only` (number, default 0) - 0 reads the embedded parameters and mesh verbatim when the file has them; non-zero (or a file without them) estimates the sources from the audio alone.
- **Returns:** `1` on success, `0` on failure.

`body_project_save(path)`

Save the parameters and sources as JSON.

- **Parameters:**
- `path` (string)
- **Returns:** `1` on success, `0` on failure.
- **Notes:** The project file also records the selected model and which parts are enabled, but not the mesh.

`body_project_load(path)`

Load a project saved with `body_project_save`.

- **Parameters:**
- `path` (string)
- **Returns:** `1` on success, `0` on failure.

Drawing

Builtin	Level	Summary
`body_draw(mode, yaw, ox, oy, oz, scale, extra)`	I	Draw the working mesh with the pen colour.
`body_draw_creases(yaw, ox, oy, oz, scale)`	A	Draw the detected crease curves.

`body_draw(mode, yaw, ox, oy, oz, scale, extra)`

Draw the working mesh with the pen colour.

- **Parameters:**
- `mode` (integer, default 0) - 0 solid triangles with smoothed normals; 1 wireframe of every `extra`-th triangle (default 1); 2 solid tinted by skin displacement, with `extra` as the full-scale push in millimetres (default 10), warm for outward and cool for inward; 3 colour by surface normal; 4 hard single-light clay shading. Modes 3 and 4 make fine relief easy to judge.
- `yaw` (number, default 0) - rotation about the vertical axis in radians.
- `ox, oy, oz` (numbers, default 0) - where the mesh origin is placed.
- `scale` (number, default 1)
- `extra` (number) - see `mode`.
- **Returns:** nothing.
- **Notes:** The mesh is y-up in metres and the renderer is y-down, so it is flipped on placement. Set the pen colour first with `set_color`.

```
GUI
bind start = do {
  open_window(640, 480)
  body_preview()
  set_color(220, 200, 180)
  body_draw(0, 0.4, 0.0, 1.0, 6.0, 1.0)
  flush_3d()
  present()
}
```

CHAPTER 26

Input, Microphone and Speech-to-Text

This chapter covers three kinds of input beyond the keyboard and mouse: game controllers, the microphone, and recognising speech in recorded audio. The keyboard and mouse builtins (`key_down`, `key_pressed`, `mouse_x`, ...) belong to the window chapter.

- **Game controllers** come in two generations of builtin. The `pad_*` family is the modern one: up to four players, edge-detected buttons, analog triggers, and the same code path on desktop and in the browser. The older `gamepad_*` family reads the first active controller by name and is the one that can make a controller vibrate. Both are backed by the `ling-input` crate ("Sensorium") and the `gilrs` library; in a browser build the `pad_*` family uses the W3C Gamepad API.
- **The microphone** (`mic_*`, plus `music_mic_pitch` in the music chapter) is backed by `ling-mic`.
- **Speech-to-text** (`stt_*`) runs the Whisper model locally. Nothing is sent over the network.

Every `pad_*`, `gamepad_*` and `mic_*` builtin is safe to call when no device exists: buttons read as not pressed, axes as 0, and the microphone as silent. Treat the return value of `mic_open` and `pad_count` as the way to find out whether hardware is present.

Game controllers: the pad_ family

Call `pad_poll` once per frame, before reading anything. It drains the controller events, advances the edge detection and returns the number of connected pads. Player slots count from 0; there are four.

Button names accepted by `pad_button` and `pad_pressed` (case-insensitive; several names are aliases of the same button):

Button	Names
bottom face	`a`, `south`, `cross`
right face	`b`, `east`, `circle`
left face	`x`, `west`, `square`

Button	Names
top face	`y`, `north`, `triangle`
left shoulder	`lb`, `l1`, `left_shoulder`
right shoulder	`rb`, `r1`, `right_shoulder`
left trigger (digital)	`lt`, `l2`, `left_trigger`
right trigger (digital)	`rt`, `r2`, `right_trigger`
start	`start`, `menu`, `options`
select	`select`, `back`, `share`, `view`
guide	`guide`, `home`
stick clicks	`l3`, `left_stick`; `r3`, `right_stick`
d-pad	`up` / `dpad_up`, `down` / `dpad_down`, `left` / `dpad_left`, `right` / `dpad_right`

An unknown name reads as not pressed. On a desktop build the face buttons are positional (the bottom button is `a` on every vendor's pad), so you never need to branch on the controller brand.

Stick direction. On desktop the vertical axes follow the screen convention: pushing a stick *up* gives a *negative* `pad_ly` or `pad_ry`. In a browser build the values are those of the Gamepad API, which use the same screen-down convention.

Builtin	Level	Summary
`pad_poll()`	B	Advance controller input by one frame.
`pad_count()`	B	Number of connected pads.
`pad_connected(player)`	B	Whether a pad is assigned to a player slot.
`pad_button(player, name)`	B	Whether a button is held down.
`pad_pressed(player, name)`	B	Whether a button went down this frame.
`pad_lx(player)`	B	Left stick, horizontal.
`pad_ly(player)`	B	Left stick, vertical.
`pad_rx(player)`	B	Right stick, horizontal.
`pad_ry(player)`	B	Right stick, vertical.
`pad_lt(player)`	I	Left analog trigger.
`pad_rt(player)`	I	Right analog trigger.
`pad_rumble(player, low, high)`	I	Request controller vibration through the Sensorium backend.

`pad_poll()`

Advance controller input by one frame.

- **Aliases:** zh `手柄轮询` · ja `パッド更新` · ko `패드폴링` · th `อัปเดตแพด`
- **Returns:** the number of connected pads (integer).
- **Notes:**
- Call it once per frame before any `pad_*` read. The first call opens the controller backend; if that fails it returns 0 and later reads give neutral values.
- Edge detection uses a fixed 60 Hz step, independent of your real frame rate.

```
GUI
bind start = do {
  open_window(640, 360)
  while window_is_open() {
    bind pads = pad_poll()
    if pad_pressed(0, "a") {
      print("A pressed on a pad; pads connected: " + str(pads))
    }
    present()
  }
}
```

`pad_count()`

Number of connected pads.

- **Aliases:** zh `手柄数` · ja `パッド数` · ko `패드수` · th `จำนวนแพด`
- **Returns:** integer, as of the last `pad_poll`.

`pad_connected(player)`

Whether a pad is assigned to a player slot.

- **Aliases:** zh `手柄连接` · ja `パッド接続` · ko `패드연결` · th `แพดเชื่อม`
- **Parameters:**
- `player` (integer, default 0) - slot 0 to 3.
- **Returns:** `true` or `false`.

`pad_button(player, name)`

Whether a button is held down.

- **Aliases:** zh `手柄按键` · ja `パッドボタン` · ko `패드버튼` · th `ปุ่มแพด`
- **Parameters:**
- `player` (integer, default 0)
- `name` (string) - see the table of names above.
- **Returns:** `true` while the button is down.

`pad_pressed(player, name)`

Whether a button went down this frame.

- **Aliases:** zh `手柄按下` · ja `パッド押下` · ko `패드눌림` · th `แพดกด`
- **Parameters:**
- `player` (integer, default 0)
- `name` (string)
- **Returns:** `true` only on the frame of the press.
- **Notes:** In a browser build only the current snapshot is available, so `pad_pressed` behaves like `pad_button` there.

`pad_lx(player)`

Left stick, horizontal.

- **Aliases:** zh `手柄左X` · ja `パッド左X` · ko `패드왼X` · th `แพดซ้ายX`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 (left) to 1 (right); `0` without a pad.
- **Notes:** The value is the stick's raw position. No extra deadzone is applied by the builtin, so a worn stick can read slightly off zero at rest; ignore values with a small absolute value in your game.

`pad_ly(player)`

Left stick, vertical.

- **Aliases:** zh `手柄左Y` · ja `パッド左Y` · ko `패드왼Y` · th `แพดซ้ายY`

- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 (up) to 1 (down); `0` without a pad.

`pad_rx(player)`

Right stick, horizontal.

- **Aliases:** zh `手柄右X` · ja `パッド右X` · ko `패드오X` · th `แพดขวาX`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 to 1.

`pad_ry(player)`

Right stick, vertical.

- **Aliases:** zh `手柄右Y` · ja `パッド右Y` · ko `패드오Y` · th `แพดขวาY`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 (up) to 1 (down).

`pad_lt(player)`

Left analog trigger.

- **Aliases:** zh `手柄左扳机` · ja `パッド左トリガー` · ko `패드왼트리거` · th `ไกแพดซ้าย`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** 0 (released) to 1 (fully pulled).

`pad_rt(player)`

Right analog trigger.

- **Aliases:** zh `手柄右扳机` · ja `パッド右トリガー` · ko `패드오트리거` · th `ไกแพดขวา`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** 0 to 1.

`pad_rumble(player, low, high)`

Request controller vibration through the Sensorium backend.

- **Aliases:** zh `手柄震动` · ja `パッド振動` · ko `패드진동` · th `แพดสั่น`
- **Parameters:**
- `player` (integer, default 0)
- `low` (number, default 0) - heavy motor, 0 to 1.
- `high` (number, default `low`) - light motor, 0 to 1.
- **Returns:** nothing.
- **Notes: Currently inert on desktop.** The gilrs backend adapter in `ling-input` deliberately ignores rumble commands, because force feedback needs effect objects that outlive the call. Use `gamepad_rumble` to make a controller vibrate. In a browser build the call does nothing as well.

```
GUI
bind start = do {
  pad_poll()
  pad_rumble(0, 0.5, 0.5)
}
```

Game controllers: the gamepad_ family

The older family addresses the first active controller (the most recently used one) by *name* rather than by player slot, and returns numbers instead of booleans. It exists mainly because it is the one that implements vibration, and because it keeps the Start button wired as Enter. Call `gamepad_poll` once per frame.

Button names for `gamepad_button`: `a`, `b`, `x`, `y`, `l1`, `r1`, `l2`, `r2`, `l3`, `r3`, `start`, `select`, `dup`, `ddown`, `dleft`, `dright`.

Axis names for `gamepad_axis`: `lx`, `ly`, `rx`, `ry` (sticks, -1 to 1) and `l2`, `r2` (triggers).

Here the vertical stick axes use the *controller's* convention, where pushing up gives a positive value: the opposite sign of `pad_ly` and `pad_ry`.

Pressing Start also counts as a press of the Enter key in `key_pressed("enter")`, so menus that wait for Enter respond to a controller with no extra code.

Builtin	Level	Summary
`gamepad_poll()`	I	Pump controller events.
`gamepad_button(name)`	I	Whether a named button is held.
`gamepad_axis(name)`	I	Value of a named axis.
`gamepad_rumble(low, high, milliseconds)`	I	Vibrate the current controller.
`gamepad_list()`	A	Describe the controllers the system can see.
`gamepad_any()`	A	Whether any button on the active controller is down.

`gamepad_poll()`

Pump controller events.

- **Aliases:** th `จอยโพล`
- **Returns:** nothing.
- **Notes:**
- Call once per frame. The most recently active controller becomes the current one.
- In a browser build `gamepad_poll` and `gamepad_rumble` do nothing; use the `pad_*` family there.

`gamepad_button(name)`

Whether a named button is held.

- **Aliases:** th `จอยปุ่ม`
- **Parameters:**
- `name` (string) - see the list above.
- **Returns:** `1` if held, otherwise `0` (also for unknown names and when no pad is connected).

`gamepad_axis(name)`

Value of a named axis.

- **Aliases:** th `จอยแกน`
- **Parameters:**
- `name` (string) - see the list above.
- **Returns:** number from -1 to 1; `0` for unknown names or no pad.

```
GUI
bind start = do {
  gamepad_poll()
  bind x = gamepad_axis("lx")
  bind y = gamepad_axis("ly")
  print(x)
  print(y)
}
```

`gamepad_rumble(low, high, milliseconds)`

Vibrate the current controller.

- **Aliases:** th `จอยสั่น`
- **Parameters:**
- `low` (number, default 0) - heavy motor, clamped to 0..1.
- `high` (number, default 0) - light motor, clamped to 0..1.
- `milliseconds` (number, default 200) - how long the vibration lasts.
- **Returns:** nothing. Nothing happens if the controller does not support force feedback.

```
GUI
bind start = do {
  gamepad_poll()
  if gamepad_button("a") == 1 {
    gamepad_rumble(0.4, 0.8, 250)
  }
}
```

`mic_open()`

Open the default microphone and start capturing.

- **Aliases:** zh `开麦克风` · ja `マイク開く` · ko `마이크열기` · th `เปิดไมค์`
- **Returns:** `1` on success, `0` if there is no device or access was refused. On the web target it returns unit.
- **Notes:** Call it once. Reading levels afterwards costs almost nothing.

```
GUI
bind start = do {
  if mic_open() == 1 {
    print("listening")
  } else {
    print("no microphone")
  }
}
```

`mic_rms()`

Loudness of the latest buffer.

- **Aliases:** zh `麦克风音量` · ja `マイクRMS` · ko `마이크RMS` · th `เสียงRMS`
- **Returns:** root-mean-square level from 0 to 1; `0` if the microphone is closed.

```
GUI
bind start = do {
  mic_open()
  bind level = mic_rms()
  print(level)
}
```

`mic_peak()`

Peak level of the latest buffer.

- **Aliases:** zh `麦克风峰值` · ja `マイクピーク` · ko `마이크피크` · th `เสียงพีค`
- **Returns:** highest absolute sample, 0 to 1; `0` if closed.

`mic_fft(bands)`

Spectrum of what the microphone hears.

- **Aliases:** zh `实时频谱` · ja `リアルタイムFFT` · ko `실시간FFT` · th `วิเคราะห์เสียงสด`
- **Parameters:**
- `bands` (integer, default 8) - number of logarithmically spaced bands from 20 Hz upward.

- **Returns:** list of `bands` numbers, bass first. Values are smoothed magnitudes. All zeros when the microphone is closed.
- **Notes:** The analyser uses a 2048-sample window and a nominal 44.1 kHz rate, and its smoothing state is shared with `music_fft`.

```
GUI
bind start = do {
  mic_open()
  bind bars = mic_fft(8)
  print(len(bars))
}
```

`stt_available()`

Whether speech recognition can start.

- **Returns:** `true` if an in-process Whisper model exists or `faster-whisper` can be imported by the Python in use.

```
GUI
bind start = do {
  if stt_available() {
    print("speech recognition ready")
  }
}
```

`stt_models()`

List the installed Whisper models.

- **Returns:** flat list `[name, size_in_megabytes, name, size_in_megabytes, ...]`, for example `small` at about 463 MB. Empty if none are found.

`stt_start(audio, model, language, words)`

Start transcribing a file.

- **Parameters:**

- `audio` (string) - path to an audio file (WAV, FLAC, OGG, MP3 or AAC; backslashes are converted to slashes).
- `model` (string, default "small") - a name from `stt_models`.
- `language` (string, default "auto") - `"auto"` to detect, or a language code such as `"en"` or `"ja"`.
- `words` (boolean or number, default true) - `false` or `0` skips word timestamps, which makes `stt_words` empty and `stt_srt` fall back to whole segments.
- **Returns:** the job id (string), or an empty string if the job could not be started.

```
GUI
bind start = do {
  bind job = stt_start("interview.wav", "small", "auto", true)
  while stt_done(job) == false {
    print("progress: " + str(stt_poll(job)))
    sleep(200)
  }
  print(stt_srt(job, 42))
}
```

`stt_poll(job)`

Progress of a job.

- **Parameters:**
- `job` (string)
- **Returns:** 0 to 1; `0` for an unknown job.

`stt_done(job)`

Whether a job has finished (successfully or not).

- **Parameters:**
- `job` (string)
- **Returns:** `true` or `false`. An unknown job counts as done.

`stt_error(job)`

Why a job failed.

- **Parameters:**
- `job` (string)
- **Returns:** an empty string while there is no error, the reason otherwise (`"cancelled"` after `stt_cancel`), and `"no such job"` for an unknown id.

`stt_language(job)`

Detected language.

- **Parameters:**
- `job` (string)
- **Returns:** language code (for example `"en"`); empty until the job has detected it.

`stt_segments(job)`

The recognised sentences so far.

- **Parameters:**
- `job` (string)
- **Returns:** flat list `[start, end, text, start, end, text, ...]`, growing as the job proceeds.

`stt_words(job)`

The recognised words so far.

- **Parameters:**
- `job` (string)
- **Returns:** flat list `[start, end, word, probability, ...]`, four entries per word; empty when `words` was false.

`stt_srt(job, max_chars)`

Subtitles in SubRip format.

- **Parameters:**
- `job` (string)
- `max_chars` (integer, default 42) - longest cue in characters. Cues break at a pause of one second or more. `0` produces one cue per whole segment.

- **Returns:** SubRip text, ready to write to a `.srt` file.

`stt_cancel(job)`

Stop a job.

- **Parameters:**
- `job` (string)
- **Returns:** `true`. The job then reports the error `"cancelled"`.

CHAPTER 27

The Graphics Crates

Everything in chapters 10 to 13 is implemented by a small family of Rust crates in the Ling repository. This chapter is for two kinds of reader: the Ling programmer who wants to know what sits underneath a builtin, and the Rust programmer who wants to embed one of these crates in another program.

For each crate you will find its purpose, its main types and functions, how Ling scripts reach it, and a worked example. The Rust examples were compiled and run against the repository as a path dependency, so their output is real. Add a crate from a checkout of the repository like this:

```
[dependencies]
ling-graphics  = { path = "path/to/ling/crates/ling-graphics" }
ling-ui        = { path = "path/to/ling/crates/ling-ui" }
ling-animation = { path = "path/to/ling/crates/ling-animation" }
ling-gpu       = { path = "path/to/ling/crates/ling-gpu" }
ling-icon      = { path = "path/to/ling/crates/ling-icon" }
```

How the pieces fit

The Ling interpreter's window, framebuffer and 3D pipeline live in the main `ling-lang` crate (`src/gfx/` and `src/runtime/mod.rs`), not in `ling-graphics`. That module has its own small camera (a yaw and a pitch stored as cosines and sines, the one `set_camera` sets), its own depth queue and software rasteriser, and its own primitive-solid builder. The graphics crates supply the parts that are shared or reusable:

Crate	What the interpreter takes from it
`ling-graphics`	The cel-shading model (`shading`) and the vector-font engine (`VectorFont`)
`ling-ui`	The `ui_*` widget geometry (`widgets`), the stroke font and hit-testing (`holo`), and the `ease` curves
`ling-animation`	The scalar drivers, easing curves and gear-train solver behind `tween_ease`, `breathe`, `spring_to` and the rest of chapter 13

Crate	What the interpreter takes from it
`ling-gpu`	Batch vertex projection for `draw_mesh` and the glTF drawing calls, and the N-body kernel used by the physics crate
`ling-icon`	Application icons for `ling build`, and the `icon_convert_to_png` builtin
`ling-wasm`	The browser entry points that run the interpreter on a web page

The rest of `ling-graphics` (scene graph, quaternion cameras, mesh builder, software renderer) and of `ling-ui` (retained widgets, menus, docking) are **Rust-only libraries**. Scripts do not reach them. They are useful on their own and are described below.

ling-graphics

Purpose. A self-contained 3D and 4D graphics library: math types, meshes, materials, a scene graph, perspective and orthographic cameras, a hyperbolic 4D camera, a software rasteriser that renders to an in-memory RGBA framebuffer, a cel-shading model, bitmap and vector fonts. It has no window system or GPU dependency and runs anywhere Rust does. Dependencies are `glam`, `serde`, `thiserror`, `fontdue` and `ttf-parser`.

Key types and functions

Module	Contents
`math`	Re-exports `glam` types `Vec2`, `Vec3`, `Vec4`, `Mat3`, `Mat4`, `Quat`, `IVec2`, `IVec3`, `UVec2`; adds `Vec4H`, `Mat5`, `Aabb`, `Frustum`, `Plane`, `Ray3`
`geometry`	`Vertex`, `Mesh`, `MeshBuilder`; generators `cube(half)`, `sphere(radius, rings, sectors)`, `icosphere(radius, subdivisions)`, `cone(radius, height, segments)`, `pyramid(base_half, height)`, `cylinder(radius, height, segments)`, `torus(major, tube, major_segs, tube_segs)`, `plane(half_size, subdivisions)`
`material`	`Material` (albedo colour and texture, normal map, metallic, roughness, emissive, alpha mode, tint, double-sided), `TextureData`, `AlphaMode`
`color`	`Color` (linear float RGBA, with constants such as `Color::RED` and constructors `from_srgb`, `from_hex`), `BlendMode`, `ColorGradient`
`scene`	`Scene` (a tree of nodes), `SceneNode`, `Transform` (translation, rotation, scale), `NodeId`; `Scene::spawn_mesh` adds a mesh node as a root
`camera`	`Camera3D` (`perspective`, `orthographic`, `look_at`, `orbit`, `unproject_ray`, `view_proj`), `Camera4D` and `HyperModel` for hyperbolic space
`renderer`	`Renderer` trait, `SoftwareRenderer`, `FrameBuffer` (RGBA bytes plus depth)

Module	Contents
`window`	`Window` : renders a `Scene` into a `FrameBuffer` (`render_scene`), presents it to the terminal as ANSI art (`present_terminal`) or runs a tick loop (`run`)
`viewport`	`Viewport` (width, height, clear colour)
`shading`	The cel and holographic lighting model: `ShadeParams` , `LightS` , `lit_vertex` , `posterize` , `posterize_soft` , `pack` and `unpack` between `0xRRGGBB` and linear floats
`vfont`	`VectorFont` : loads a TTF or OTF, caches glyph outlines on disk, `measure` , `advance` , `glyph_outline` , `has_glyph`
`font`	Bitmap fonts: `FontAtlas` , `GlyphFont` , `generate_text_mesh` , `measure_text`
`animation`	`EaseFunction` , `Lerp` , `Keyframe` , `Track` , `Timeline` (a graphics-side twin of the `ling-animation` types)

Conventions differ from the Ling builtins. The Rust library uses a right-handed system with the camera looking along -**Z** and +Y up, and Rust `f32` throughout. The Ling builtins use +Y down and look along +Z (chapter 10). Do not mix their numbers.

How Ling exposes it

- `set_shade_mode` , `set_cel_bands` , `set_shadow_color` , `set_rim` and `set_ambient` edit the `ShadeParams` the interpreter passes to `shading::lit_vertex` for every solid. The defaults match `ShadeParams::default()` : 4 bands, ambient 0.22, shadow `(0.10, 0.13, 0.30)` , rim 0.6 in `(0.45, 0.85, 1.0)` , holographic sheen on.
- `font_load` , `font_width` , `font_advance` and `font_glyph_outline` wrap `VectorFont` (`from_path_weight` , `measure` , `advance` and `glyph_outline`). The on-disk glyph cache is `cache/fonts/<file-stem>[@<weight>]/<codepoint>.ling` .

Worked example: rendering a sphere in memory

```
use ling_graphics::*;
use std::sync::Arc;

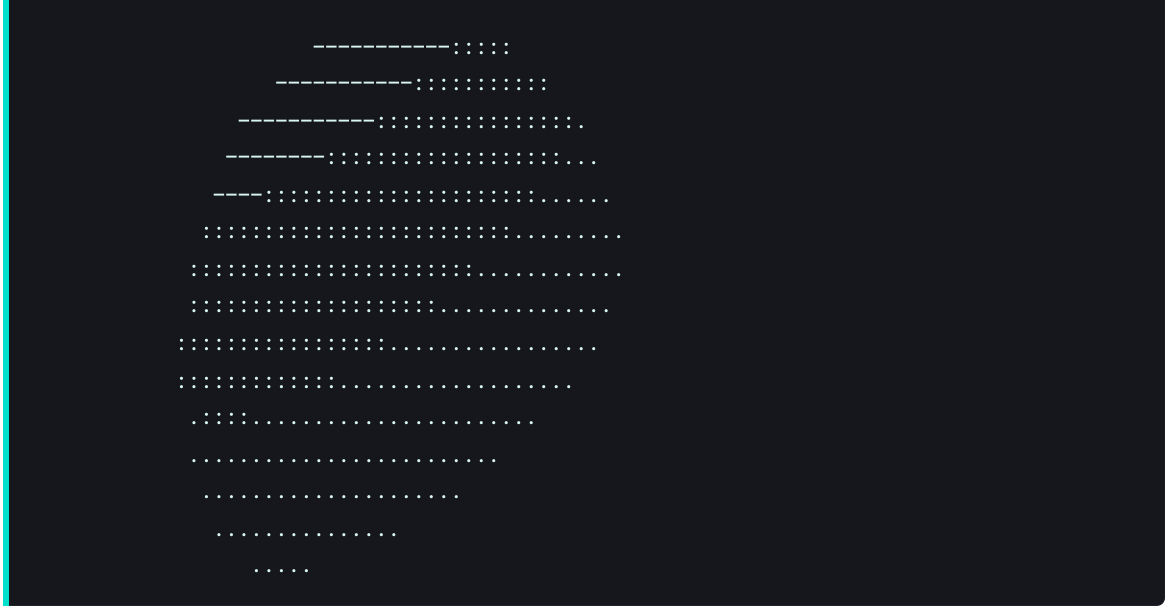
fn main() {
    // 1. A scene: one sphere, one material, one transform.
    let mut scene = Scene::new();
    let mesh = Arc::new(geometry::sphere(0.8, 16, 24));
    let material = Arc::new(Material::new(Color::from_srgb(230, 120, 90)));
    let place = scene::Transform::from_translation(Vec3::new(0.0, 0.0, -2.2));
    scene.spawn_mesh("ball", mesh, material, place);

    // 2. A camera at the origin looking down -Z, 60 degrees field of view.
    let camera = Camera3D::perspective(60.0, 60.0 / 28.0 * 0.5, 0.1, 100.0);

    // 3. Render to an RGBA framebuffer in memory.
    let mut window = window::Window::new(viewport::Viewport::new(60, 28,
Color::BLACK));
    let fb = window.render_scene(&scene, &camera);
    println!("framebuffer {}x{}, {} bytes", fb.width, fb.height, fb.pixels.len());

    // 4. Show it as ASCII art by pixel luminance.
    let peak = fb.pixels.chunks(4).map(|p|
p[0].max(p[1]).max(p[2])).max().unwrap_or(1).max(1) as f32;
    let ramp: Vec<char> = " .:-=+*#%@".chars().collect();
    for y in 0..fb.height as usize {
        let mut row = String::new();
        for x in 0..fb.width as usize {
            let i = (y * fb.width as usize + x) * 4;
            let lum = (0.2126 * fb.pixels[i] as f32 + 0.7152 * fb.pixels[i + 1] as f32
                + 0.0722 * fb.pixels[i + 2] as f32) / peak;
            row.push(ramp[(lum.clamp(0.0, 1.0) * (ramp.len() - 1) as f32) as usize]);
        }
        println!("{row}");
    }
}
```

```
OUTPUT
framebuffer 60x28, 6720 bytes
```



The 28 rows of the output are shown with trailing blank rows removed. The sphere is lit from the upper left, so its lit side is the brighter `-` and `:` and its shaded side the dim `.`. Terminal characters are about twice as tall as wide, which is why the camera aspect is halved.

Worked example: fonts and shading from Rust

The same engine as `font_width` and the cel-shading builtins:

```
use ling_graphics::{shading, VectorFont};

fn main() {
    let mut font = VectorFont::from_path_weight("assets/fonts/Exo2.ttf",
Some(600.0)).unwrap();
    println!("width of 'Hello Ling' at 32 px: {:.3}", font.measure("Hello Ling",
32.0));
    println!("advance of 'H' (em): {:.3}", font.advance('H'));
    let outline = font.glyph_outline('I', 0.05);
    println!("'I' has {} contour(s); advance {:.3}", outline.polylines.len(),
outline.advance);

    let mut params = shading::ShadeParams::default();
    params.rim = 0.0;                       // no rim glow
    params.holo = false;                    // no iridescent sheen
    params.shadow = [0.05, 0.05, 0.08];     // near-neutral shadow tint
    let light = shading::LightS { pos: [0.0, -5.0, 0.0], color: [1.0, 1.0, 1.0],
intensity: 0.6, radius: 0.0 };
    let base = shading::unpack(0xE6785A);
    let facing = shading::lit_vertex(base, [0.0, -1.0, 0.0], [0.0, 0.0, 0.0], [0.0,
0.0, -5.0], &[light], &params);
    let away = shading::lit_vertex(base, [0.0, 1.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0,
-5.0], &[light], &params);
    println!("lit  {:06x}", shading::pack(facing));
    println!("dark {:06x}", shading::pack(away));
    println!("posterised to 3 bands: {:06x}", shading::pack(shading::posterize(facing,
3)));
}
```

OUTPUT

```
width of 'Hello Ling' at 32 px: 147.712
advance of 'H' (em): 0.681
'I' has 1 contour(s); advance 0.279
lit  c66c59
dark 58322e
posterised to 3 bands: be6755
```

The first line is the same number the Ling program in chapter 12 printed for `font_width(f, 32, "Hello Ling")`. Running this example from a directory creates a `cache/fonts/Exo2@600/` folder holding the glyph files it extracted.

---

ling-ui

Purpose. A user-interface library with two halves. The first is a **retained-mode widget set and application controls** for tools and editors: widgets and views, a flex-style layout helper, context menus and menu bars, split panes, a property inspector (drag-value fields, toggles, dropdowns, collapsible sections), a keyboard shortcut registry, a command palette, tooltips, spring and tween animation, and a software renderer stub. The second is the **2030 holographic vector toolkit** used by the `ui_*` builtins: pure functions that return resolved vector geometry for meters, gauges and controls.

Default feature `layout` pulls in the `taffy` flexbox engine; the optional `gpu` feature adds `wgpu`.

Key types and functions

Area	Items
Widgets	`Widget` trait (`id` , `render` , `handle`), `View` , `Label` , `Button` , `Event` and `MouseButton`
Layout	`LayoutNode` , `Style` , `FlexDirection` , `simple_layout(parent, count, style)`
Menus	`MenuBar` , `MenuCategory` , `MenuItem` (`action` , `checkable` , `submenu` , `separator`), `MenuPopup`
Docking	`SplitPane` , `SplitDirection` (`compute_layout` , `update_drag`)
Inspector	`ValueDrag` , `Toggle` , `Dropdown` , `PropertySection`
Commands	`ShortcutRegistry` , `KeyCombo` , `Modifiers` , `CommandPalette` , `CommandItem`
Tooltip	`Tooltip` (hover delay and card placement)
Animation	`Easing` , `ease(curve, a, b, t)` , `Spring` , `Tween`
Vector widgets	`widgets::*` : `bar` , `segbar` , `gauge` , `ring` , `vu` , `spark` , `battery` , `button` , `toggle` , `slider` , `checkbox` , `tabs` , `progress` , `tooltip` , `stepper` , `healthbar` , `cooldown` , `counter` , `minimap` , `dpad` , `slotgrid` , `vignette` , `gauge3d` , `panel3d` , `radar3d` , `radar` , `compass` , `reticle` , `target` , `panel` , `scanlines` ; the result type is `widgets::Draw`
Holo	`holo::hit_rect` , `glyph` , `text_lines` , `text_width` , `corner_brackets` , `beveled_rect`
Rendering	`SoftwareRenderer` (a pixel buffer with `clear` , `render_widget` , `pixels`)

The vector widgets are **pure**: each takes the geometry and the live state (hover, value, active) as arguments, never reads the mouse and returns a `Draw`, a list of `fills` and `strokes` as `(0xRRGGBB colour, Vec<[x, y]> points)`. The host rasterises them however it likes. Screen coordinates are y-down with the origin at the top-left, like the Ling builtins.

How Ling exposes it

- Every `ui_*` builtin of chapter 12 calls the matching function in `ling_ui::widgets` (for example `ui_bar` calls `widgets::bar`) and rasterises the returned `Draw` with antialiased fills and strokes into the framebuffer. The interpreter, not the crate, reads the mouse and passes `hover` and `down` in.
- `ui_text` and `ui_hot` use `holo::text_lines` and `holo::hit_rect`.
- `ease(name, t)` uses `ling_ui::Easing::from_name(name).apply(t)`.
- The retained-mode widgets, menus, docking, shortcuts and command palette are **not** reachable from Ling scripts.

Worked example

```
use ling_ui::*;

fn main() {
    // Widgets return resolved vector geometry (no drawing, no input handling).
    let bar = widgets::bar(20.0, 20.0, 200.0, 14.0, 0.7, 0x00d2ff, 0x2c3e64);
    println!("bar: {} fills, {} strokes", bar.fills.len(), bar.strokes.len());
    for (colour, poly) in &bar.fills { println!("  fill {:06x} with {} points", colour,
poly.len()); }

    // Flex-style layout of equal children.
    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} y {:.0} w {:.0} h {:.0}", n.x, n.y, n.width,
n.height);
    }

    // Animation helpers.
    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());
    let mut tween = Tween::new(10.0, 20.0, 0.5, Easing::CubicOut);
    tween.update(0.25);
    println!("tween halfway in time: {:.2}", tween.value());

    // Command palette filtering.
    let items = vec![
        CommandItem::new("file.open", "Open File", "File", Some("Ctrl+O")),
        CommandItem::new("file.save", "Save File", "File", Some("Ctrl+S")),
        CommandItem::new("view.zoom", "Zoom In", "View", None::<String>),
    ];
    let mut palette = CommandPalette::new(items);
    palette.open();
    palette.query = "save".into();
    let hits: Vec<_> = palette.filtered_items().iter().map(|i|
```

```
i.label.clone()).collect();
    println!("palette 'save' -> {:?}, confirm -> {:?}", hits,
palette.confirm_selected());
}
```

```
OUTPUT
bar: 1 fills, 1 strokes
  fill 00d2ff with 5 points
child 0: x 5 y 5 w 90 h 90
child 1: x 105 y 5 w 90 h 90
child 2: x 205 y 5 w 90 h 90
spring after 1 s: 1.000 (settled: true)
tween halfway in time: 18.75
palette 'save' -> ["Save File"], confirm -> Some("file.save")
```

The same bar from a Ling script is `ui_bar(20, 20, 200, 14, 70, 100)`, covered with all the other widgets in chapter 12.

---

ling-animation

Purpose. "Anima", Ling's unified animation system: one rig, any target, any temperament. It provides easing curves and a `Lerp` trait, keyframe tracks and a looping timeline, a joint hierarchy with world and skinning matrices, the organic-to-mechanical **temperament** value, scalar drivers, mechanisms and a facial-expression rig. It has no graphics dependency and is safe for WebAssembly.

Key types and functions

Module	Contents
`ease`	`EaseFunction` (21 curves: `Linear`, `Step`, the `In`, `Out` and `InOut` forms of quad, cubic, sine and expo, `ElasticIn` / `ElasticOut`, `BackIn` / `BackOut` / `BackInOut`, `BounceIn` / `BounceOut`) with `from_name` and `apply`; `Lerp` (implemented for `f32`, `f64`, `Vec2`, `Vec3`, `Vec4`, `Quat`); `tween_ease(a, b, t, ease)`
`track`	`Keyframe<T>`, `Track<T>` (`key(time, value, ease)`, `sample(t)`, `duration()`), `Timeline` (`play`, `pause`, `stop`, `tick(dt)`, `normalized()`; fields `time`, `speed`, `playing`, `looping`, `duration`)

Module	Contents
`rig`	`Transform`, `Joint` (name, parent, rest and local transforms, temperament, limits), `Rig` (`add`, `find`, `set_local`, `reset`, `resolve`, `world`, `world_pos`, `skinning_matrices`)
`temperament`	`Temperament` (`ORGANIC` 0, `HYBRID` 0.5, `MECHANICAL` 1; `blend`, `liveliness`, `rigidity`)
`scalar`	`breathe`, `wobble`, `idle_noise`, `gait_phase`, `gait_swing`, `gait_lift`, `spring_step`, `two_bone_ik`, `two_bone_fk`, `gear`, `cam_lift`, `piston`, `rack`
`mechanism`	`gear_train(angle, &teeth)`, `four_bar(crank_angle, ground, crank, coupler, rocker)`
`creature`	`biped_legs`, `quadruped_legs` (each leg gives `(swing, lift)`), `breathing_scale`
`face`	`FaceRig`, `FacePoint`, `Region`, `Expression` (`Blink`, `Smile`, `Frown`, `Surprise`, `Squint`, `JawOpen`, `Anger`, `Sad`) with `solve`, `solve_point` and `eye_aperture_l`
top level	`Animator`: a handle-indexed owner of rigs and timelines (`add_rig`, `add_timeline`, `tick`, `tick_all`)

How Ling exposes it

The `scalar` and `mechanism::gear_train` functions, plus `EaseFunction` through `tween_ease`, are the Ling builtins of chapter 13: `breathe`, `wobble`, `gait_phase`, `gait_swing`, `gait_lift`, `spring_to` (which wraps `spring_step`), `ik2` (`two_bone_ik`), `gear_couple` (`gear`), `gear_train`, `cam_lift`, `piston`, `rack` and `tween_ease`. Tracks, timelines, rigs, temperaments, creatures, `four_bar` and the face rig have **no** script builtins yet.

Worked example

```
use ling_animation::{scalar, mechanism, creature, EaseFunction, Timeline, Track,
FaceRig, Expression, Temperament};

fn main() {
    // Keyframe track: 0 -> 100 -> 40 over two seconds, with easing per key.
    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..5 {
        let t = clock.tick(0.5);
        println!("t = {:.1}s  value = {:.2}", t, track.sample(t).unwrap());
    }

    // Procedural gait: two legs in anti-phase (swing, lift).
    let [left, right] = creature::biped_legs(0.25, 1.0, 30.0, 10.0);
    println!("left  swing {:.1} lift {:.1}", left.0, left.1);
    println!("right swing {:.1} lift {:.1}", right.0, right.1);

    // Mechanisms.
    println!("gear train {:?}", mechanism::gear_train(1.0, &[12.0, 36.0, 12.0]));
    println!("four-bar rocker angle {:?}", mechanism::four_bar(0.8, 4.0, 1.0, 3.5,
3.0).map(|a| (a * 1000.0).round() / 1000.0));
    println!("breathing {:.3}", scalar::breathe(0.25, 1.0, 0.2));

    // Temperament blend.
    let t = Temperament::new(0.25);
    println!("temperament {:.2}: liveliness {:.2} rigidity {:.2} blend(10, 20) =
{:.1}",
        t.value(), t.liveliness(), t.rigidity(), t.blend(10.0, 20.0));

    // Facial expression: a half smile.
    let face = FaceRig::new();
    let rest = face.rest("mouth_corner_l").unwrap();
    let smiled = face.solve_point("mouth_corner_l", &[(Expression::Smile,
0.5)]).unwrap();
```

```
    println!("mouth corner rest ({:.2}, {:.2}) -> half smile ({:.2}, {:.2})", rest.x,
rest.y, smiled.x, smiled.y);
}
```

```
OUTPUT
t = 0.5s  value = 50.00
t = 1.0s  value = 100.00
t = 1.5s  value = 47.50
t = 0.0s  value = 0.00
t = 0.5s  value = 50.00
left  swing 30.0 lift 10.0
right swing -30.0 lift 0.0
gear train [1.0, -0.33333334, 1.0]
four-bar rocker angle Some(-2.2)
breathing 1.200
temperament 0.25: liveliness 0.75 rigidity 0.25 blend(10, 20) = 12.5
mouth corner rest (-0.38, -0.62) -> half smile (-0.42, -0.55)
```

Notes on the output. The timeline loops, so the fourth tick wraps from 2.0 seconds back to 0.0. The face rig works in a normalised face space with y up, x to the right and the origin between the eyes, which is why the mouth corner sits at negative y and a smile raises it (the y value increases toward zero). The first four-bar value is an angle in radians for the "open" branch, or `None` if the linkage cannot close at that crank angle.

---

ling-gpu

Purpose. Data-parallel compute with a transparent fallback: the rest of the engine asks for `ling_gpu::backend()` and gets a CUDA backend when the `cuda` feature is on and an NVIDIA device is present, and otherwise a multi-threaded CPU backend (via `rayon`). Callers never branch on the device. The default build is pure Rust and runs everywhere.

Key types and functions

Item	Purpose
`backend()`	The process-wide best backend (created once)

Item	Purpose
`device_name()`, `gpu_active()`, `gpu_count()`	Report the device, whether a GPU is in use and how many GPUs
`trait Backend`	`name`, `is_gpu`, `gpu_count`, `nbody_accel(pos, mass, g, soften, out)`, `project_points(world, camera, out)`
`CameraParams`	The interpreter camera's numbers: yaw and pitch as cosine and sine, screen centre, focal length, `zdist` and position

Both operations use interleaved `x, y, z` floats. `project_points` writes `(screen_x, screen_y, camera_depth)` per vertex, with the same formula as the Ling camera in chapter 10. `nbody_accel` computes gravitational or charge acceleration for every body in O(n squared).

How Ling exposes it

- `draw_mesh` and the glTF drawing calls project every vertex of a mesh in one `project_points` call.
- The physics crate uses `nbody_accel`.
- The Ling language has no builtin to query the backend; build the interpreter with `--features cuda` to enable the GPU path.

Worked example

```
fn main() {
    println!("device: {} (gpu active: {}, gpus: {})",
        ling_gpu::device_name(), ling_gpu::gpu_active(), ling_gpu::gpu_count());
    let cam = ling_gpu::CameraParams { cry: 1.0, sry: 0.0, crx: 1.0, srx: 0.0,
        cx: 320.0, cy: 200.0, focal: 400.0, zdist: 5.0, tx: 0.0, ty: 0.0, tz: 0.0 };
    let world = [0.0f32, 0.0, 2.0,  1.0, 1.0, 2.0];
    let mut out = [0.0f32; 6];
    ling_gpu::backend().project_points(&world, &cam, &mut out);
    println!("projected: {:?}", out);
    let pos = [0.0f32, 0.0, 0.0,  1.0, 0.0, 0.0];
    let mut acc = [0.0f32; 6];
    ling_gpu::backend().nbody_accel(&pos, &[1.0, 1.0], 1.0, 0.0, &mut acc);
    println!("n-body accel: {:?}", acc);
}
```

```
OUTPUT
device: CPU (gpu active: false, gpus: 0)
projected: [320.0, 200.0, 2.0, 377.14285, 257.14285, 2.0]
n-body accel: [1.0, 0.0, 0.0, -1.0, 0.0, 0.0]
```

The projected point `(1, 1, 2)` lands at `(377.14, 257.14)`: `320 + 400 * 1 / (2 + 5)`. Two unit masses one unit apart pull toward each other with acceleration 1.

---

ling-icon

Purpose. Turns an icon source (`.svg`, `.png`, `.jpg`, `.bmp` or `.ico`) into a multi-resolution Windows `.ico` and embeds it in an executable from a build script. SVG is rendered faithfully (gradients, filters, blur) through `resvg`; raster images are resized with the `image` crate; an existing `.ico` passes through unchanged.

Key functions

Function	Purpose
`ico_bytes(src, sizes)`	Build `.ico` bytes from any supported source
`write_ico(src, out, sizes)`	The same, written to a file
`png_bytes(src, size)`	One square PNG of `size` by `size` pixels, never touching Windows resources
`embed_in_build(ico_path)`	From a build script, embed the icon in every binary of the crate (Windows only; a no-op elsewhere)
`rasterize_and_embed(src, out_dir)`	Rasterise to `<stem>.ico` and embed it in one step
`DEFAULT_SIZES`	`[16, 32, 48, 64, 128, 256]`

Sizes are clamped to 1 through 256 for `.ico`.

How Ling exposes it

- `ling build` uses the crate for the application icon. Name an icon with `--icon <file.svg|png|ico>` or with an `icon = "images/icon.svg"` line in the project's `Ling.toml`. With neither, a bundled default is used. This applies to Windows builds.

- `icon_convert_to_png(src, out, size)` is a builtin that converts any supported icon source to a PNG. It returns `true` on success and `false` on failure.

```
bind start = do {
    bind ok = icon_convert_to_png("ic.svg", "ic_out.png", 64)
    println("ok = " + str(ok))
    println("exists = " + str(file_exists("ic_out.png")))
}
```

```
OUTPUT
ok = true
exists = true
```

(Run in a folder that contains a small `ic.svg`. The builtin is present in the standard `ling` build; it depends on the interpreter's `web` feature.)

Worked example (Rust)

```
use std::path::Path;

fn main() {
    let svg = std::env::temp_dir().join("demo-icon.svg");
    std::fs::write(&svg, r##"<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 100
100"><rect width="100" height="100" rx="20" fill="#2244ff"/><circle cx="50" cy="50"
r="28" fill="#ffcc00"/></svg>"##).unwrap();
    let png = ling_icon::png_bytes(Path::new(&svg), 64).unwrap();
    println!("png: {} bytes, signature {:?}", png.len(), &png[1..4]);
    let ico = ling_icon::ico_bytes(Path::new(&svg), ling_icon::DEFAULT_SIZES).unwrap();
    println!("ico: {} bytes, {} images", ico.len(), u16::from_le_bytes([ico[4],
ico[5]]));
}
```

```
OUTPUT
png: 1773 bytes, signature [80, 78, 71]
ico: 16140 bytes, 6 images
```

The signature bytes 80, 78, 71 are the letters `PNG`. The `.ico` holds six images, one per default size.

ling-wasm

Purpose. The browser entry points for the Ling interpreter. It is a `cdylib` built with `wasm-bindgen` around the main `ling-lang` crate: it hands the interpreter an `OffscreenCanvas` to draw on, accepts keyboard, mouse and gamepad events from JavaScript, and exposes the Web Audio hooks. The interpreter renders the same software framebuffer as on the desktop and uploads each presented frame to the canvas through WebGL2.

Exported JavaScript functions

Function	Purpose
`init_canvas(canvas)`	Give the runtime an `OffscreenCanvas`; call once before `run_program`
`register_module(path, source)`	Provide the source of a module so that `use "path"` resolves in the browser; call for every module first
`run_program(source)`	Parse and run a Ling program; errors go to the browser console
`on_key_down(key)`, `on_key_up(key)`	Keyboard events, with a `KeyboardEvent.key` string
`clear_frame_keys()`	Clear the per-frame "pressed" state; call at the start of each frame
`on_mouse_move(x, y)`	Pointer position in canvas pixels
`on_mouse_button(button, pressed, x, y)`	Button 0 left, 1 middle, 2 right (the numbering of a DOM `MouseEvent`)
`on_gamepad_button(index, button, pressed, value)`, `on_gamepad_axis(index, axis, value)`, `on_gamepad_connected(index, name)`, `on_gamepad_disconnected(index)`	Gamepad state from the browser's Gamepad API
`audio_ready()`	Resume the Web Audio context; must be called from a user-gesture handler
`play_audio_buffer(pcm, sample_rate)`, `set_master_volume(v)`	Queue PCM audio, set the master volume (0 to 1)

How Ling exposes it

A Ling program targeted at the browser is an ordinary program using the builtins of chapters 10 to 13. Differences to know about on the web:

- `font_load` always returns `-1` (the web runtime does not read font files); fall back to `ui_text`.
- Audio extras such as `ui_sound` and `audio_blip` are silent stubs.
- Key names sent through `on_key_down` are normalised by the runtime: a space becomes `"space"`, `ArrowUp` and its relatives become `"up"`, `"down"`, `"left"` and `"right"`, `Enter`, `Escape`, `Tab` and `Backspace` become their lower-case names, the modifier keys become `"shift"`, `"ctrl"` and `"alt"` (with no left and right variants, so `key_down("lshift")` does not match on the web), and any other key is lower-cased. Letters and digits therefore work with the same names as on the desktop.
- `window_is_open()` is always true, and `open_window` only sets the canvas size.

Worked example: a page that runs a Ling program

The repository's `examples-web/gfx/` folder is a working page. The main page creates a worker and transfers the canvas to it; the worker loads the wasm module, then runs the program. This is the worker, reduced to its essentials:

```
// worker.js: runs the Ling interpreter inside a Web Worker.
importScripts('./ling_wasm.js');

wasm_bindgen('./ling_wasm_bg.wasm').then(() => {
    // both calls are valid only after the wasm module has been instantiated
    wasm_bindgen.init_canvas(pendingCanvas);      // an OffscreenCanvas from the page
    wasm_bindgen.run_program(pendingSource);      // the text of a .ling file
});
```

The page side, from the same example:

```
const canvas = document.getElementById('canvas');
const worker = new Worker('worker.js');
const offscreen = canvas.transferControlToOffscreen();
worker.postMessage({ type: 'init', canvas: offscreen }, [offscreen]);
worker.postMessage({ type: 'run', source });   // source is the program text
```

These snippets were taken from the repository's example and could not be run in a browser for this book, so treat the event-forwarding calls (`on_key_down` and the rest) as checked against the crate's source and type declarations only. The checked-in package was built in the "no modules" style, which exposes everything under a global `wasm_bindgen` object, as above.

CHAPTER 28

The Audio, Physics and Input Crates

The builtins described in the previous chapters are thin layers over ordinary Rust crates. Each crate is published on crates.io and can be used on its own in a Cargo project, with no Ling interpreter involved. This chapter is a field guide to nine of them: what each is for, its main types and functions, which Ling builtins lead to it, and a short worked example in Rust that was compiled and run.

Crate	Version	What it is	Reached from Ling through	Native dependencies
`ling-audio`	2030.0.13	Positional audio engine, FFT analyser, colour palettes	`audio_*`, `ui_sound`, `morph_note`, `music_fft`, `mic_fft`	`cpal`, `hound`, `rustfft`
`ling-music`	2030.0.9	Decode, analyse and synthesise music; MIDI, lyrics, rhythm judging	`music_*`	`symphonia`, `midly`, `cpal`, `rustfft`
`ling-daw`	2030.0.2	The LingStationX engine: instruments, effects, mixer, project files	`daw_*`, `fs_*`, `dlg_*`	`cpal`, `resvg`, `rfd`, `midir`, `ureq` and more
`ling-osdaw`	2030.0.2	Freestanding tracker sequencer and synthesiser	not reachable from scripts (used by the LingOS kernel)	`libm` only
`ling-physics`	2030.1.18	Soft and rigid bodies, liquid, terrain, weather, hyperbolic geometry	`soft_*`, `rb_*`, `liquid_*`, `gltf_*`, camera warp	`glam`, `rayon`, `ling-gpu`
`ling-game`	2030.1.9	Dialog boxes, ECS, 2D physics, procedural meshes and textures	`dialog_*` only	`ling-core`, `ling-runtime`
`ling-body`	2030.0.2	Acoustic body toolkit: humanoid SDF, wave solver, mesh and WAV codecs	`body_*`	`rayon`, `rustfft`, `serde_json`, `ling-music`
`ling-input`	2030.0.9	Sensorium: gamepads, touch, gestures, XR, action maps	`pad_*`	`glam`, optional `gilrs`
`ling-mic`	2030.0.12	Microphone capture	`mic_*`, `music_mic_pitch`	`cpal`, `hound`

All nine are dual-licensed `Apache-2.0 OR MIT`, share the version scheme `2030.x.y`, and were built and run for this chapter with Rust 1.98 on Windows. The crates that open audio devices use `cpal`, which on Linux needs the ALSA development package (for example `libasound2-`

`dev` on Debian and Ubuntu).

Adding a crate to a Cargo project

From crates.io, name the crate and a version in `Cargo.toml`:

```
[dependencies]
ling-music = "2030.0.9"
ling-physics = "2030.1.18"
```

From a checkout of the Ling repository, use a path dependency. The crates read dependency versions from the workspace of the repository, which Cargo resolves for you, so a path dependency on a single crate works without copying the whole workspace:

```
[dependencies]
ling-music = { path = "../ling/crates/ling-music" }
```

Optional features are switched on in the same table, for example `ling-input = { version = "2030.0.9", features = ["gilrs"] }` for native gamepads, or `ling-physics = { version = "2030.1.18", features = ["cuda"] }` for the GPU fast path.

ling-audio

Purpose. Real-time audio synthesis with positional sound, and the analysis tools behind visualisers.

Key types.

- `AudioEngine`: opens the default output device (`AudioEngine::new()` returns an error when there is none) and runs the mixer on the device thread. All methods take `&self`, so share it freely.
- Tones: `set_tone(slot, ToneParams)`, `clear_tone(slot)`.
- One-shots: `blip(freq, amp, dur, Wave)`, `sfx(x, y, z, w, freq, amp, dur, Wave)`, `morph_note(...)`, `stop_all_sfx()`.
- Samples: `add_sample(mono, rate) -> id`, `play_sample(...)`, `play_sample_flat(...)`, `set_sample_voice(...)`, `stop_sample(voice)`, `sample_duration(id)`.
- Master chain: `fx_delay`, `fx_reverb`, `fx_lowpass`, `set_master_volume`.

- Listener and background music: `set_listener`, `set_listener_pos`, `load_bgm`, `set_bgm_volume`.
- `ToneParams { x, y, z, w, freq, amp, lfo_rate, lfo_depth }` with sensible defaults.
- `Wave`: `Sine`, `Square`, `Saw`, `Triangle`, `Noise`, and `Wave::from_name`.
- `FftAnalyzer`: `new(fft_size, sample_rate)`, `push_samples`, `freq_bands(n)`, `dominant_freq`, `rms`, `is_beat`, `beat_ratio`, `set_window(Window)` with `Hann`, `Hamming`, `Blackman` or `Rectangular`.
- `CosPalette` (`rainbow`, `fire`, `ocean`, `psychedelic`, `frequency`), plus `freq_texture` and `waveform_texture`, which turn a spectrum or a waveform into RGBA pixels.

From Ling. The interpreter creates one `AudioEngine` at start-up and every `audio_*` builtin forwards to it. `music_fft` and `mic_fft` feed an internal `FftAnalyzer`.

Rust example. The analyser needs no audio device, so this program runs anywhere. The `play` function shows the engine calls; it is compiled but not called here.

```rust
use ling_audio::{AudioEngine, FftAnalyzer, ToneParams, Wave};

// Needs an output device, so it is only compiled in this check, never called.
#[allow(dead_code)]
fn play() -> Result<(), Box<dyn std::error::Error>> {
    let audio = AudioEngine::new()?;
    audio.set_master_volume(0.6);
    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);
    audio.fx_reverb(0.3);
    Ok(())
}

fn main() {
    // The analyser needs no device: feed it a 1 kHz sine and read the spectrum back.
    let mut fft = FftAnalyzer::new(2048, 44_100);
    let sine: Vec<f32> = (0..2048)
        .map(|i| (2.0 * std::f32::consts::PI * 1000.0 * i as f32 / 44_100.0).sin())
        .collect();
    fft.push_samples(&sine);
    println!("dominant frequency: {:.0} Hz", fft.dominant_freq());
    println!("bands: {}", fft.freq_bands(8).len());
}
```

```
OUTPUT
dominant frequency: 991 Hz
bands: 8
```

The 991 Hz is the frequency of the FFT bin nearest 1000 Hz. Bins are spaced by the sample rate divided by the window size, here about 21.5 Hz.

The same sound from Ling.

A positional tone and a blip (needs an audio device)

```
GUI
bind start = do {
  audio_volume(0.6)
  audio_tone(0, 2.0, 0.0, 6.0, 1.0, 196.0, 0.12, 0.5, 0.02)
  audio_blip(880.0, 0.06, "square", 0.25)
  audio_fx_reverb(0.3)
}
```

ling-music

Purpose. Everything about music as data and sound: decoding, tempo and key analysis, pitch detection, a General-MIDI-style synthesiser driven by small patch files, MIDI files, `.lrc` lyrics and rhythm-game judging.

Key modules and types.

- `decode`: `load(path)` and `from_bytes(&[u8])` return a `DecodedAudio { stereo, mono, rate, channels, duration }` for WAV, FLAC, OGG Vorbis, MP3 and AAC.
- `analysis`: `onsets`, `bpm`, `beat_grid`, `chroma`, `key`, `key_name`, `fft_bands_at_pos`, `fft_groups_at_pos`, `rms`.
- `pitch`: `detect(samples, rate)` and `detect_range`, a normalised-autocorrelation fundamental detector returning `Option<f32>`.
- `note`: `midi_to_hz`, `hz_to_midi`, `name_to_midi`, `parse_pitch`, `midi_to_name`, `hz_to_name`, `scale_steps`.
- `synth` and `patch`: `Synth::new(rate)`, `add_patch`, `note_on`, `note`, `note_off_pitch`, `next_sample`; `patch::from_str` and `patch::from_path` parse the patch text format described in the music chapter. `Patch`, `Op`, `Adsr`, `Filter`, `Lfo` and `Wave` are public.
- `midi`: `load`, `from_bytes` return a `MidiSong { notes, duration }` whose `MidiNote` has `time`, `dur`, `midi`, `vel` and `channel`.
- `karaoke`: `Lyrics::parse`, `line_at`, `index_at`, and `pitch_score`.
- `rhythm`: `Grade::judge(delta_ms)`, `Beatmap::from_beats`, and `Scorer` with combo and accuracy.
- `MusicEngine`: opens its own output stream and mixes one decoded track with the synthesiser: `set_track`, `play`, `pause`, `stop`, `seek`, `position`, `set_volume`, `add_patch`, `note`, `note_on`, `note_off`.

From Ling. `music_load` is `decode::load`, `music_bpm` is `analysis::bpm`, `music_note` goes to `MusicEngine::note`, and so on; every `music_*` builtin is one of the calls above.

Rust example. Synthesise a note offline, then listen to it with the pitch detector. No audio device is involved.

```rust
use ling_music::{note, patch, pitch, Synth};

fn main() {
    // A one-operator sine patch, written in the same text format as a .ling patch file.
    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 piano = synth.add_patch(patch::from_str(text));

    // Play A4 for half a second and pull 0.5 s of mono samples out of the synth.
    synth.note(piano, 69, 0.9, 0.5);
    let samples: Vec<f32> = (0..22_050).map(|_| synth.next_sample()).collect();

    // Detect the pitch of a window in the middle of the note.
    let hz = pitch::detect(&samples[4_000..8_096], 44_100).expect("pitch");
    println!("detected {:.1} Hz = {}", hz, note::hz_to_name(hz));
    println!("A4 is {} Hz", note::midi_to_hz(69.0));
    println!("C4 parses to MIDI {:?}", note::name_to_midi("C4"));
}
```

```
OUTPUT
detected 440.0 Hz = A4
A4 is 440 Hz
C4 parses to MIDI Some(60)
```

The same helpers from Ling.

Pitch helpers

```
bind start = do {
  bind a4 = music_hz("A4")
  print(a4)
  print(music_note_name(a4 * 1.02))
  print(round(music_pitch_score(a4 * 1.02, a4) * 100) / 100)
}
```

```
OUTPUT
440
A4
0.66
```

ling-daw

Purpose. The audio engine of the LingStationX workstation: a project model with tracks, MIDI and audio clips, ten built-in instrument kinds plus SoundFont playback, a library of effects, routing, automation, tempo maps, offline rendering to WAV, FLAC and Ogg, MIDI and tracker import, and a large set of music-theory, vocal and song-generation tools. It is by far the largest crate of the group, with tens of thousands of lines.

Key types.

- `Daw`: the scripting facade. `Daw::new()` starts an audio output (it carries on without one) and `daw.call(name, &[Val]) -> Option<Result<Val, String>>` runs any `daw_*`, `fs_*` or `dlg_*` builtin. `None` means the name is not one of ours.
- `Val`: the value type crossing the boundary: `Num(f64)`, `Str(String)`, `List(Vec<Val>)` and `Unit`.
- `Engine` / `SharedEngine`: the real-time engine behind a project (`daw.engine()`), for hosts that want to drive playback and rendering themselves.
- Modules such as `project` (save and load), `export`, `midi_import`, `theory`, `songgen` and `dsp` (instruments and effects) hold the implementation.

From Ling. The interpreter keeps one `Daw` and forwards every builtin whose name begins with `daw_`, `fs_` or `dlg_` to `Daw::call`; the shared `st_*` and `px_*` helpers live in the interpreter itself.

Rust example. The program below does exactly what a script does: it builds a project and renders it to a WAV file without touching the audio device. It was built with `ling-daw = "2030.0.2"` alone; the first build takes a few minutes because the crate pulls in many libraries.

```
use ling_daw::{Daw, Val};

/// Call one `daw_*` builtin by name, exactly as a Ling script would.
fn call(daw: &mut Daw, name: &str, args: &[Val]) -> Val {
    daw.call(name, args)
        .expect("not a daw builtin")
        .unwrap_or_else(|e| panic!("{name}: {e}"))
}

fn main() {
    let mut daw = Daw::new();

    call(&mut daw, "daw_new", &[]);
    call(&mut daw, "daw_set_bpm", &[Val::Num(100.0)]);
    let track = call(&mut daw, "daw_track_add", &[Val::Str("synth".into())]);
    let Val::Num(t) = track else { panic!("expected a track index") };

    // A clip at beat 0, four beats long, with two notes in it.
    let clip = call(&mut daw, "daw_clip_add_midi", &[Val::Num(t), Val::Num(0.0),
Val::Num(4.0)]);
    let Val::Num(c) = clip else { panic!("expected a clip index") };
    for (beat, key) in [(0.0, 60.0), (1.0, 64.0)] {
        call(
            &mut daw,
            "daw_note_add",
            &[Val::Num(t), Val::Num(c), Val::Num(beat), Val::Num(0.75), Val::Num(key),
Val::Num(0.8)],
        );
    }

    // Render offline to a WAV file: no audio device is needed for this.
    let ok = call(
        &mut daw,
        "daw_bounce",
        &[Val::Str("demo.wav".into()), Val::Num(0.0), Val::Num(4.0), Val::Num(0.5),
Val::Num(16.0), Val::Str("wav".into()), Val::Num(0.5)],
    );
    println!("bounce returned {ok:?}");
```

```
    println!("status: {:?}", call(&mut daw, "daw_status", &[]));
    println!("notes: {:?}", call(&mut daw, "daw_clip_notes", &[Val::Num(t),
Val::Num(c)]));
}
```

```
OUTPUT
bounce returned Num(1.0)
status: Str("Rendered demo.wav")
notes: List([Num(0.0), Num(0.75), Num(60.0), Num(0.800000011920929), Num(0.0),
Num(0.5), Num(0.5), Num(1.0), Num(0.75), Num(64.0), Num(0.800000011920929), Num(0.0),
Num(0.5), Num(0.5)])
```

The Ling script that does the same is the offline session in the DAW section of the audio chapter.

ling-osdaw

Purpose. "LingStation for LingOS": a tracker-style sequencer and synthesiser that needs no operating system. It is `#![no_std]` with `alloc`, and its only dependency is `libm`. The LingOS kernel feeds `Engine::render` into the AC'97 sound mixer. The crate is also an ordinary library on a desktop, which makes it a pleasant way to generate chiptune-like audio with no audio stack.

Key types and constants.

- `Song`: an 8-channel by 64-row grid of `Cell { note, vel }` (note 0 is empty, 1 to 127 a MIDI key, 255 a note-off), eight `Channel { inst, vol, pan, mute }`, plus `bpm`, `rpb` (rows per beat), `length` and four master effect levels in `fx` (drive, echo, reverb, crusher). `Song::empty()`, `Song::demo()`, `to_bytes()`, `from_bytes()`, `seconds()` and `tail_seconds()`.
- `Engine`: `Engine::new(song)`, `play`, `stop`, `seek(row)`, `trigger(channel, key, vel)`, `release(channel)`, `all_off`, `render(&mut [i16])` for interleaved stereo at `SAMPLE_RATE` (48000 Hz), and `Engine::render_wav(&Song) -> Vec<u8>`.
- Constants: `CHANNELS` = 8, `MAX_ROWS` = 64, `VOICES` = 24, and `INSTRUMENTS`, the fourteen built-in instrument names (Saw Lead, Square Bass, Pluck, Pad, FM Bell, Kick, Snare, Hat, Supersaw, Growl, Voice, Organ, Clap, Tom).
- `script`: pure edits of a song, each returning the number of cells it changed: `arpeggiate`, `humanize`, `reverse`, `euclid`, `transpose`, `transpose_melodic`, `repeat_half`, `snap_to_scale`, `detect_key`.

- `mix::auto_mix`, `midi::to_midi` and `midi::from_midi`, and `fx::Chain`.

From Ling. It is not reachable from scripts. The LingOS kernel is its only client inside the repository; ordinary scripts use `ling-daw` through the `daw_*` builtins.

Rust example.

```
use ling_osdaw::{script, Cell, Engine, Song};

fn main() {
    // Start from the built-in demo groove and edit it with a note script.
    let mut song = Song::demo();
    let changed = script::transpose(&mut song, Some(2), 12);
    println!("transposed {changed} cells up an octave on channel 2");

    // Put one extra note on the grid and render the whole song to a 16-bit WAV in
memory.
    song.cells[0][4] = Cell { note: 72, vel: 12 };
    println!("{:.1} s at {} BPM", song.seconds(), song.bpm);
    let wav = Engine::render_wav(&song);
    println!("WAV: {} bytes, starts with {:?}", wav.len(),
std::str::from_utf8(&wav[..4]).unwrap());
}
```

```
OUTPUT
transposed 16 cells up an octave on channel 2
3.9 s at 122 BPM
WAV: 1811452 bytes, starts with "RIFF"
```

ling-physics

Purpose. Simulation and world building: forces, vectors, rigid bodies, soft bodies, a 2D liquid, procedural terrain, trees and grass, weather and a day-night cycle, glTF models, and the maths of hyperbolic space. It re-exports `glam`'s `Vec2`, `Vec3`, `Vec4`, `Quat` and `Mat4`. Features: `with-gltf` (glTF loading through the `gltf` crate; without it `GltfModel::load` returns an error; the Ling interpreter enables it) and `cuda` (GPU fast path for batched operations such as n-body).

Key modules and types.

- `rigid`: `RigidBody` (position, velocity, mass, restitution, friction, half extents, orientation, angular velocity), `PhysicsWorld { bodies, gravity }` with `add` and `step`, `resolve_collision`. The default gravity is `(0, -9.81, 0)`: this world is y-up.
- `soft`: `SoftBody::sphere(centre, radius, rings, sectors, mass)`, `integrate(dt, gravity, substeps)`, `floor_collision`, `contain`, `kick`, `spin`, `centroid`, `deformation`, `angular_speed`. Its floor convention is y-down, like the Ling renderer.
- `liquid`: `LiquidGrid::new(w, h)`, `splat(x, y, kind, amount, radius)` with `WATER` and `OIL`, `set_gravity`, `step`, `mix_amount`, `sample_rgb`, and `step_all(&mut [LiquidGrid], dt)` which runs grids in parallel.
- `forces`: `gravity_force`, `drag_force`, `spring_force`, `buoyancy_force`, `wind_force`, `nbody_accel` and about two dozen more, many with `_4d` variants; `vector`: vector helpers including 4D versions.
- `hyperbolic`: Poincare-ball operations `lambda`, `distance`, `mobius_add`, `exp_map`, `log_map`, `parallel_transport`; `HyperbolicSphereWorld { radius, curvature, gravity }` with `gravity_dir`, `to_poincare`, `from_poincare`, `world_distance`, `up_at`; and the pentagonal tiling helpers `pentagon_vertices`, `translate_pentagon`, `pentagon_tiling`.
- `terrain` (`fbm`, `TerrainConfig`, `TerrainManager` with chunked level of detail), `weather` (`WorldTime`, `Weather`, `Atmosphere` with sun and moon direction and sky colour), `foliage` (`TreeParams::generate`, `gen_grass`), `world` (`WorldDescriptor`, `WorldState`, rooms and portals) and `gltf` (`GltfModel`).

From Ling. `soft_*`, `rb_*` and `liquid_*` wrap `soft`, `rigid` and `liquid` one for one; the `gltf_*` builtins wrap `GltfModel`; `set_camera_hyper` and `tm_screen_lens` implement the hyperbolic projection in the renderer using the same formulas. The terrain, weather, foliage and `hyperbolic` modules have no script builtins of their own.

Rust example.

```
use ling_physics::hyperbolic::{distance, HyperbolicSphereWorld};
use ling_physics::liquid::{LiquidGrid, OIL, WATER};
use ling_physics::rigid::{PhysicsWorld, RigidBody};
use ling_physics::soft::SoftBody;
use ling_physics::Vec3;

fn main() {
    // Rigid body: a box dropped from 5 m onto a floor at y = 0 (y is up here).
    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);
    }
    println!("box rests at y = {:.2}", world.bodies[id].pos.y);

    // Soft body: a squashy ball on a floor (here y is down, so the floor is the
largest y).
    let mut ball = SoftBody::sphere(Vec3::new(0.0, -3.0, 0.0), 1.0, 8, 12, 1.0);
    for _ in 0..300 {
        ball.integrate(0.016, Vec3::new(0.0, 9.8, 0.0), 4);
        ball.floor_collision(2.0, 0.5);
    }
    println!("ball nodes: {}, centre y = {:.2}", ball.nodes.len(), ball.centroid().y);

    // Liquid: water and oil start apart.
    let mut pool = LiquidGrid::new(16, 16);
    pool.splat(8.0, 4.0, OIL, 1.0, 3.0);
    pool.splat(8.0, 12.0, WATER, 1.0, 3.0);
    println!("mix before: {:.3}", pool.mix_amount());

    // Hyperbolic distance grows without bound as points approach the boundary.
    let d1 = distance(Vec3::ZERO, Vec3::new(0.5, 0.0, 0.0));
    let d2 = distance(Vec3::ZERO, Vec3::new(0.99, 0.0, 0.0));
    println!("d(0, 0.5) = {d1:.3}, d(0, 0.99) = {d2:.3}");
    let w = HyperbolicSphereWorld::default();
```

```
    println!("gravity at the pole: {:?}", w.gravity_force(Vec3::new(0.0, 50.0, 0.0),
1.0));
}
```

```
OUTPUT
box rests at y = 0.50
ball nodes: 86, centre y = 1.09
mix before: 0.000
d(0, 0.5) = 1.099, d(0, 0.99) = 5.293
gravity at the pole: Vec3(0.0, 9.81, 0.0)
```

Notice the two different conventions in one program: the box falls toward negative y, the ball toward positive y. The same rule applies to the `rb_*` and `soft_*` builtins.

The rigid-body half from Ling.

A box dropped onto a floor

```
bind start = do {
  bind box = rb_add(0.0, 5.0, 0.0, 1.0)
  bind frame = 0
  while frame < 240 {
    rb_step(0.016)
    rb_floor(box, 0.0, 0.5, 0.6)
    bind frame = frame + 1
  }
  bind p = rb_pos(box)
  print("box rests at y = " + str(round(p[1] * 100) / 100))
}
```

```
OUTPUT
box rests at y = 0.5
```

ling-game

Purpose. Building blocks for games: an entity-component store, a minimal 2D physics module, procedural meshes and textures, and the typed-out dialog box. Optional features (`bevy`, `physics2d`, `physics3d`) integrate with the Bevy engine and the Avian physics library.

Key types.

- `Entity` and `ComponentStore<T>`: `Entity::spawn` and `despawn` allocate ids (reusing freed ones); `ComponentStore<T>` is a sparse-set store with `insert`, `get`, `get_mut`, `remove` and `iter`. One store per component type is the whole ECS.
- `GameApp` and `GameSystem`: a fixed-rate loop that calls `update(dt)` on every registered system until `quit`.
- `physics`: `Vec2`, `RigidBody` (force, integrate), `Aabb` with `overlaps`.
- `mesh`: `ProceduralMesh { vertices, indices }` with generators `sphere`, `torus`, `cylinder`, `grid`, `cube`, `mobius`, `icosphere`, `hyperbolic_patch`, `pentagon` and `merge`. A vertex is `[x, y, z, nx, ny, nz, u, v]`.
- `texture`: `Palette` (`rainbow`, `fire`, `ocean`, `psychedelic`, `neon`, `forest`) and image generators returning RGBA bytes: `checkerboard`, `noise`, `gradient`, `mandelbrot`, `julia`, `voronoi`, `hypnotic_spiral`, `halftone`, `ripple`, `freq_map` and more.
- `Dialog`: the typed-out dialog described in the physics chapter. `Dialog::new(text, chars_per_second)`, `update(dt)`, `advance()`, `is_typing()`, `is_closed()`, `visible_runs()`. Pure logic: it never draws.
- `AudioMixer` and `SoundHandle`: a placeholder handle allocator.

From Ling. Only `Dialog` is reachable: the `dialog_*` builtins are a thin wrapper, and `dialog_draw` renders the runs that `visible_runs` returns. The ECS, `GameApp`, 2D physics, meshes and textures are for Rust programs; scripts do their own state handling (see the `st_*` store for a shared dictionary).

Rust example.

```
use ling_game::entity::{ComponentStore, Entity};
use ling_game::physics::{Aabb, RigidBody, Vec2};
use ling_game::{mesh, Dialog};

struct Health(i32);

fn main() {
    // ECS: allocate entities and attach components stored in sparse sets.
    let mut ids = Entity::new();
    let mut health: ComponentStore<Health> = ComponentStore::new();
    let hero = ids.spawn();
    let slime = ids.spawn();
    health.insert(hero, Health(10));
    health.insert(slime, Health(3));
    health.get_mut(slime).unwrap().0 -= 3;
    println!("hero {} hp, slime {} hp", health.get(hero).unwrap().0,
health.get(slime).unwrap().0);

    // 2D physics and collision helpers.
    let mut body = RigidBody::new(0.0, 0.0, 2.0);
    body.apply_force(Vec2::new(4.0, 0.0), 1.0);
    body.integrate(1.0);
    println!("body moved to x = {}", body.pos.x);
    println!("overlap: {}", Aabb::new(0.0, 0.0, 1.0, 1.0).overlaps(Aabb::new(1.5, 0.0,
1.0, 1.0)));

    // Procedural mesh.
    let ball = mesh::sphere(1.0, 8, 12);
    println!("sphere: {} vertices, {} triangles", ball.vertices.len(),
ball.indices.len() / 3);

    // Typed-out dialog.
    let mut d = Dialog::new("Hi {n}Ling{/}||Bye", 100.0);
    d.update(0.5);
    println!("typing: {}", d.is_typing());
}
```

```
OUTPUT
hero 10 hp, slime 0 hp
body moved to x = 2
overlap: true
sphere: 117 vertices, 192 triangles
typing: false
```

Note that despawning an entity returns its id to the allocator but does not clear its components; call `remove` on each store yourself.

The dialog half from Ling.

The same dialog from a script

```
bind start = do {
  dialog_show("Hi {n}Ling{/}|Bye", 100)
  dialog_step(0.5)
  print("typing: " + str(dialog_typing()))
}
```

```
OUTPUT
typing: false
```

ling-body

Purpose. The acoustic body toolkit described in the physics chapter: a parametric humanoid or imported model is voxelised to a signed distance field, a finite-difference wave solver excites it, a level-set membrane follows the wave and the surface is extracted as a mesh. Meshes export as OBJ, STL and GLB (with a fitted skeleton); the whole recipe round-trips through WAV files.

Key types and modules.

- `BodyKit`: the facade with everything a front end needs: `param_set`, `source_add`, `preview_rest`, `sim_begin`, `sim_advance(budget_ms)`, `sim_done`, `mesh_from_sim`, `mesh_info`, `crease_detect` and `crease_apply`, `listen_begin`, `export_mesh`, `wav_export`, `wav_import`, `project_json`.
- `Params`, `Source`, `Waveform`: the parameter table (47 named numbers) and the acoustic source list. `Mesh` and `Tris`: triangle data. `Grid`: the voxel lattice.

- Lower-level modules: `sdf` (distance fields from triangles), `wave` (the solver), `membrane`, `mesher` (surface extraction and decimation), `humanoid` (the joint hierarchy), `model` (glTF, `.lmesh` and OBJ import), `codec` and `wavio` (the WAV container with embedded recipe), `curves` and `detail` (creases), `signature` (the listening pass), `analysis` (estimating sources from audio).

From Ling. Every `body_*` builtin is a method call on a single `BodyKit` inside the interpreter.

Rust example.

```rust
use ling_body::BodyKit;

fn main() {
    let mut kit = BodyKit::new();
    kit.param_set("sr", 8000.0);
    kit.param_set("steps", 60.0);
    kit.param_set("seal", 1.0);
    kit.source_add([0.0, 1.0, 0.0], 0, 220.0);

    let total = kit.sim_begin().expect("sim");
    println!("steps: {total}");
    while !kit.sim_done() {
        kit.sim_advance(40.0); // work for about 40 ms, then return
    }
    kit.mesh_from_sim().expect("mesh");
    let (verts, tris, watertight) = kit.mesh_info();
    println!("{verts} vertices, {tris} triangles, watertight: {watertight}");
    println!("{}", kit.export_mesh("body.obj").expect("export"));
}
```

```
OUTPUT
steps: 60
5006 vertices, 10008 triangles, watertight: true
10008 triangles → body.obj
```

The Ling version of this program is the worked example in the body section of the physics chapter.

ling-input

Purpose. "Sensorium", a unified input layer. Every device (gamepad, flight stick, touch screen, on-screen virtual controls, VR hand, gaze ray) becomes events, and games read remappable *actions* without naming a device.

Key types.

- `Sensorium::new(max_players)`: the hub. Per frame call `begin_frame`, then `pump(&mut backend)` (or `ingest(event)` by hand), then `update(dt)`. Read `player(slot) -> Option<&Gamepad>`, `keyboard`, `mouse`, `touch`, `gestures`, `onscreen`, `motion`, `xr` and `actions`.
- `Gamepad`: `is_down`, `just_pressed`, `just_released`, `axis`, `stick`, `dpad`, plus the `left_stick`, `right_stick`, `left_trigger` and `right_trigger` fields. Buttons are positional (`GamepadButton::South` and so on, 23 of them including paddles and touchpad), and `Layout` maps them to the printed glyphs of Xbox, PlayStation, Nintendo or generic pads.
- `Button` (edge detection with auto-repeat), `Axis`, `Stick` and `Curve` (deadzone, saturation, response curves).
- `ActionMap`, `Action`, `Binding`, `ActionSet` and `ActionSets`: named actions bound to buttons, keys or stick directions, with a stack of contexts (gameplay, menu) of which the top one can block those below.
- `Rumble`, `HapticEffect`, `HapticPlayer`, `TriggerEffect`: force-feedback commands and time-sampled effects.
- `TouchPool`, `GestureRecognizer`, `Gesture` (tap, long-press, swipe, pinch, rotate), `OnScreenControls`, `VirtualStick`, `VirtualButton`, `VirtualDpad` and `ControlTheme`.
- `Xr`, `Headset`, `XrController`, `HandSkeleton`, `Gaze`: VR poses.
- `InputBackend` (trait) with `ManualBackend` for tests and replays, and `GilrsBackend` behind the `gilrs` feature for native controllers. Input is `serde`-serialisable by default for replays and rollback netcode.

From Ling. The interpreter creates a `Sensorium` with four player slots and a `GilrsBackend` on the first `pad_poll`. The `pad_*` builtins read `player(i)`. Note that the gilrs adapter does not implement rumble, which is why `pad_rumble` has no effect and `gamepad_rumble` (a separate code path in the runtime) is the way to vibrate a controller. The older `gamepad_*` builtins use `gilrs` directly and do not go through `Sensorium`.

Rust example. Events are pushed in by hand with `ingest`, so the program is deterministic and needs no controller.

```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
```

With the `gilrs` feature enabled, a real game loop replaces `ingest` with `pump`:

```
use ling_input::backend::GilrsBackend;
use ling_input::{GamepadButton, Sensorium};

fn game_loop() -> Result<(), Box<dyn std::error::Error>> {
    let mut sensorium = Sensorium::new(4);
    let mut backend = GilrsBackend::new()?;
    loop {
        sensorium.begin_frame();
        sensorium.pump(&mut backend);
        sensorium.update(1.0 / 60.0);
        if let Some(pad) = sensorium.player(0) {
            if pad.just_pressed(GamepadButton::South) {
                println!("jump! stick = {:?}", pad.left_stick);
            }
        }
        std::thread::sleep(std::time::Duration::from_millis(16));
    }
}
```

The matching Ling loop.

Reading a pad in a game loop (needs a controller)

```
GUI
bind start = do {
  open_window(640, 360)
  while window_is_open() {
    pad_poll()
    if pad_pressed(0, "a") {
      print("jump! stick x = " + str(pad_lx(0)))
    }
    present()
  }
}
```

ling-mic

Purpose. Cross-platform microphone capture. Native builds use `cpal`. The WebAssembly build is a stub: `MicInput::open` fails with `MicError::Unsupported` and capture must be done in JavaScript.

Key types.

- `MicConfig { sample_rate, channels, buffer_frames, device }`. The default is 44,100 Hz mono with 512-frame buffers on the default device. `input_devices()` lists device names (empty on the web).
- `MicInput`: `open(config)`, `start(callback)` with the callback receiving each buffer of `f32` samples, `start_framed(callback)` (the callback also receives the channel count), `stop`, and meters `peak()`, `rms()`, `latest_samples()` (a rolling mono window for FFT or pitch work), `sample_rate()`, `channels()` and `device_name()`.
- `MicError`: the error type.

From Ling. `mic_open` calls `MicInput::open` with the default configuration and starts a stream with an empty callback, because scripts poll instead of receiving callbacks: `mic_rms` and `mic_peak` read the meters, `mic_fft` and `music_mic_pitch` read `latest_samples`.

Rust example. Compiled but not run here, since running it would open the real microphone; it lists the devices.

```
use ling_mic::{input_devices, MicConfig, MicInput};

// Compiled only: running it would open the real microphone.
#[allow(dead_code)]
fn listen() -> Result<(), Box<dyn std::error::Error>> {
    let mic = MicInput::open(MicConfig { sample_rate: 48_000, ..MicConfig::default()
})?;
    mic.start(|samples| {
        let rms = (samples.iter().map(|s| s * s).sum::<f32>() / samples.len() as
f32).sqrt();
        println!("level {rms:.3}");
    })?;
    std::thread::sleep(std::time::Duration::from_secs(3));
    println!("peak {:.3}, rms {:.3}", mic.peak(), mic.rms());
    mic.stop();
    Ok(())
}

fn main() {
    println!("{} input device(s) found", input_devices().len());
}
```

The callback runs on the audio thread: keep it short and do not block. The number of devices printed depends on the machine.

The microphone from Ling (needs a microphone)

```
GUI
bind start = do {
  if mic_open() == 1 {
    bind level = mic_rms()
    print("level: " + str(level))
  }
}
```

PART

Showcase

Real applications written entirely in Ling.

CHAPTER 29

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

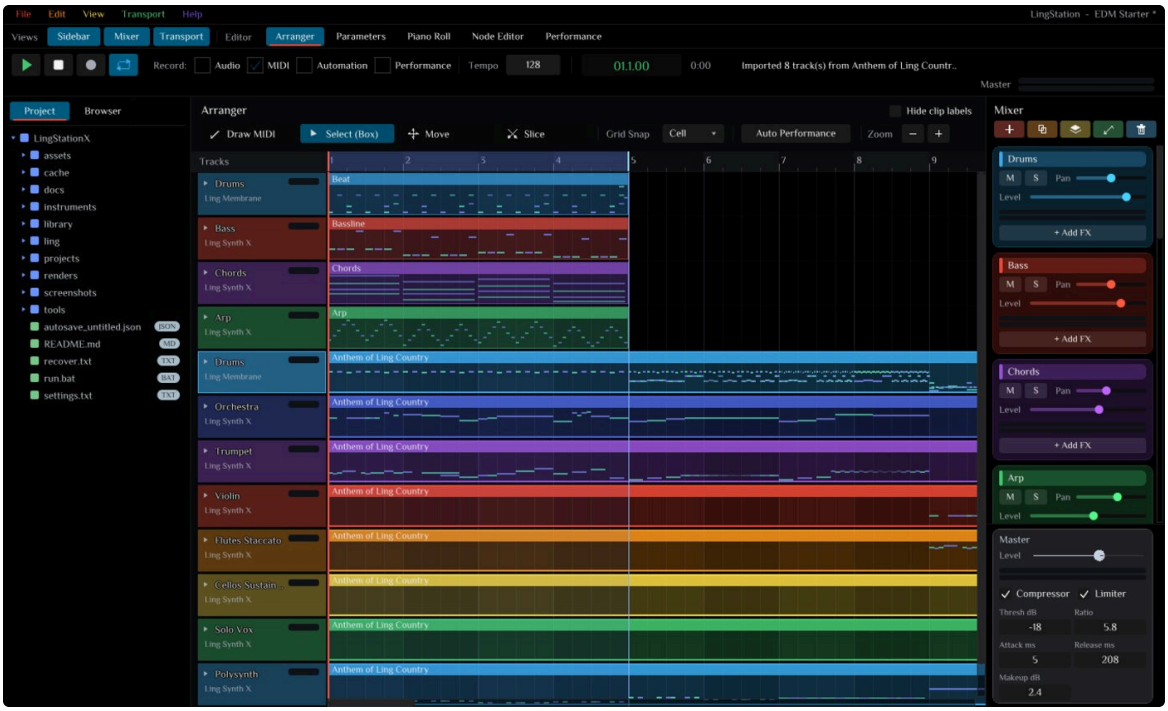


Figure 90.1. The arranger. Each track tile, clip, ruler tick and mixer strip is drawn by Ling code every frame; the clips of an imported eight-track song sit below four template tracks.

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.

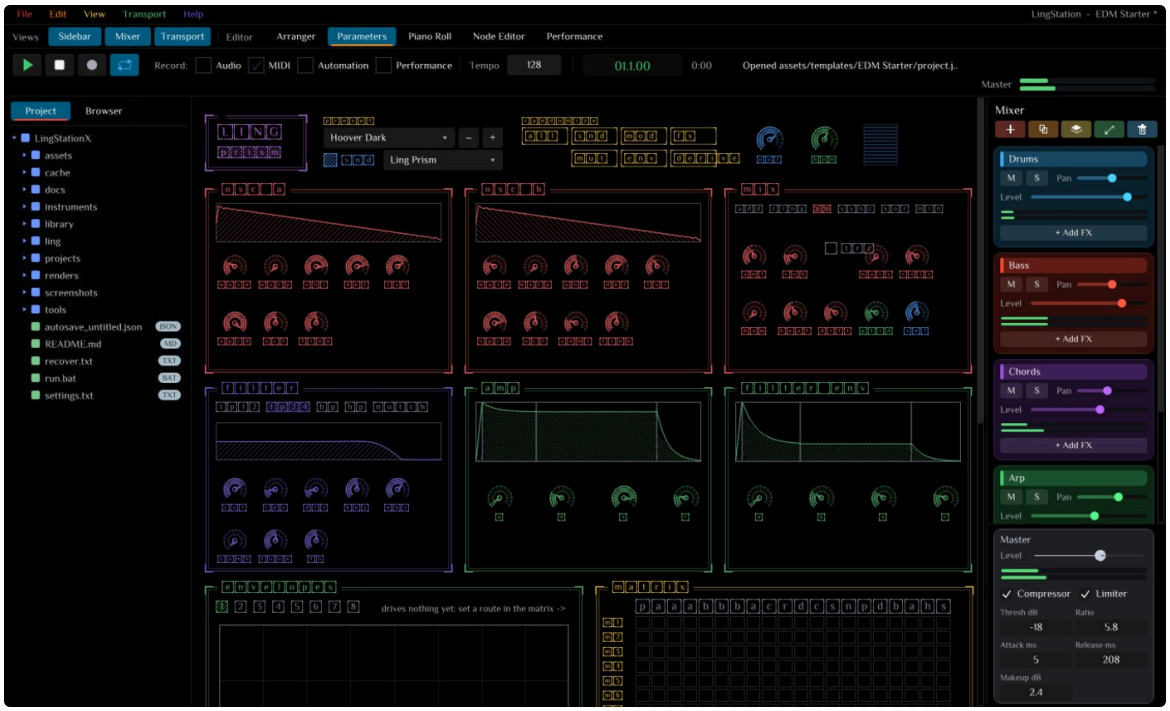


Figure 90.2. The Parameters tab showing Ling Prism, a 16-voice synthesiser, with its oscillators, filter, envelopes and modulation matrix. The panel is a Ling function reading the instrument's parameter list from the engine.

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

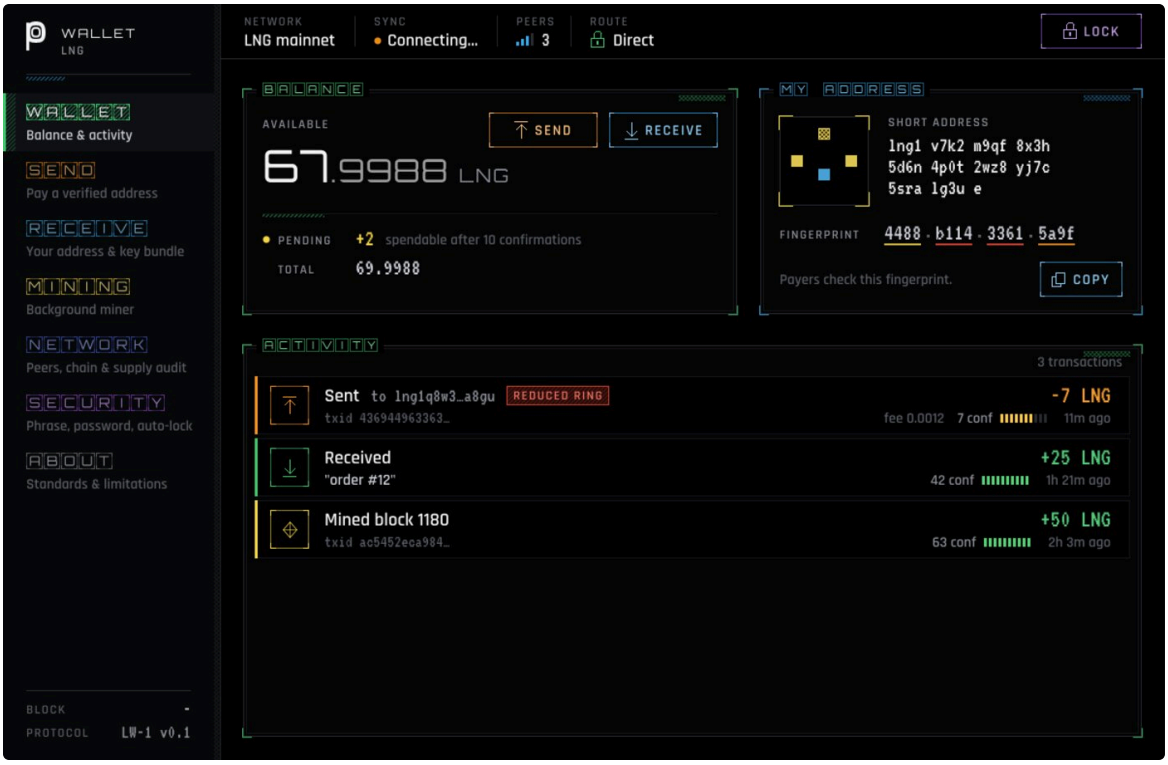


Figure 90.3. The wallet screen on the simulated backend: balance, the fingerprint of the user's address and three activity rows, including a send flagged as using a reduced ring.

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.

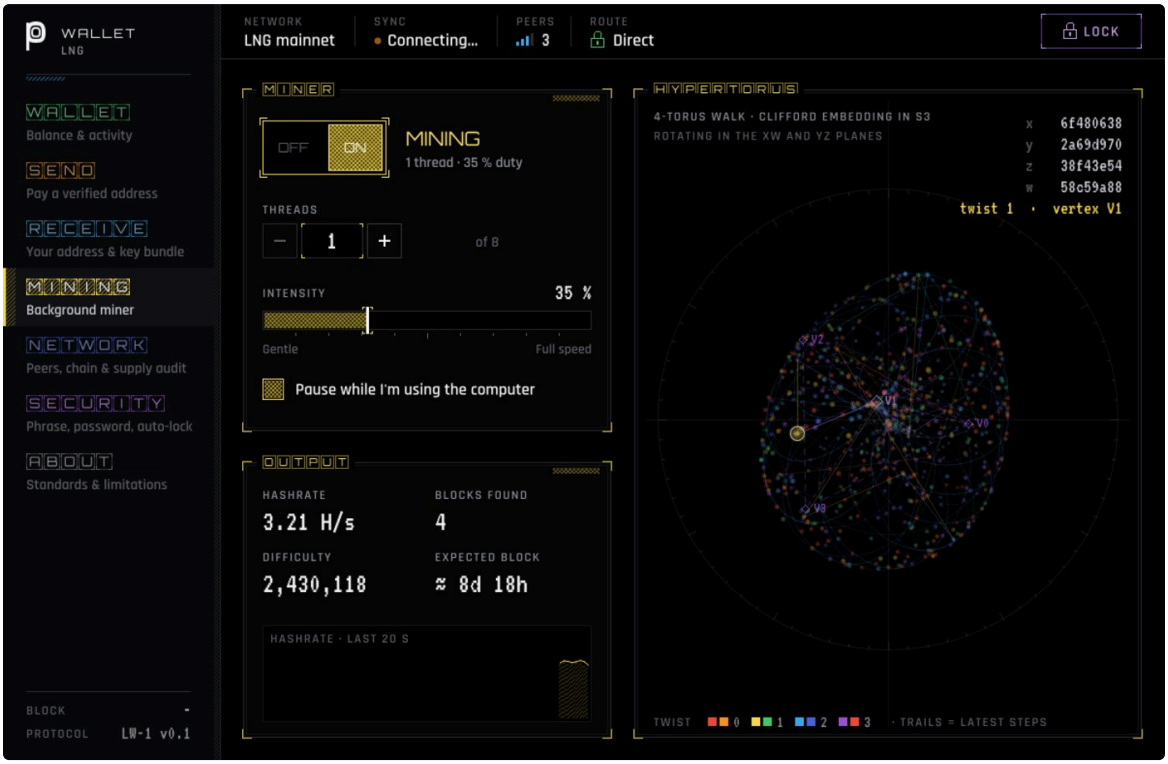


Figure 90.4. The Mining screen on simulated data: thread count, duty-cycle intensity, a pause-while-I-am-using-the-computer option and the 4-D torus walk drawn from `lw_pow_trace` samples.

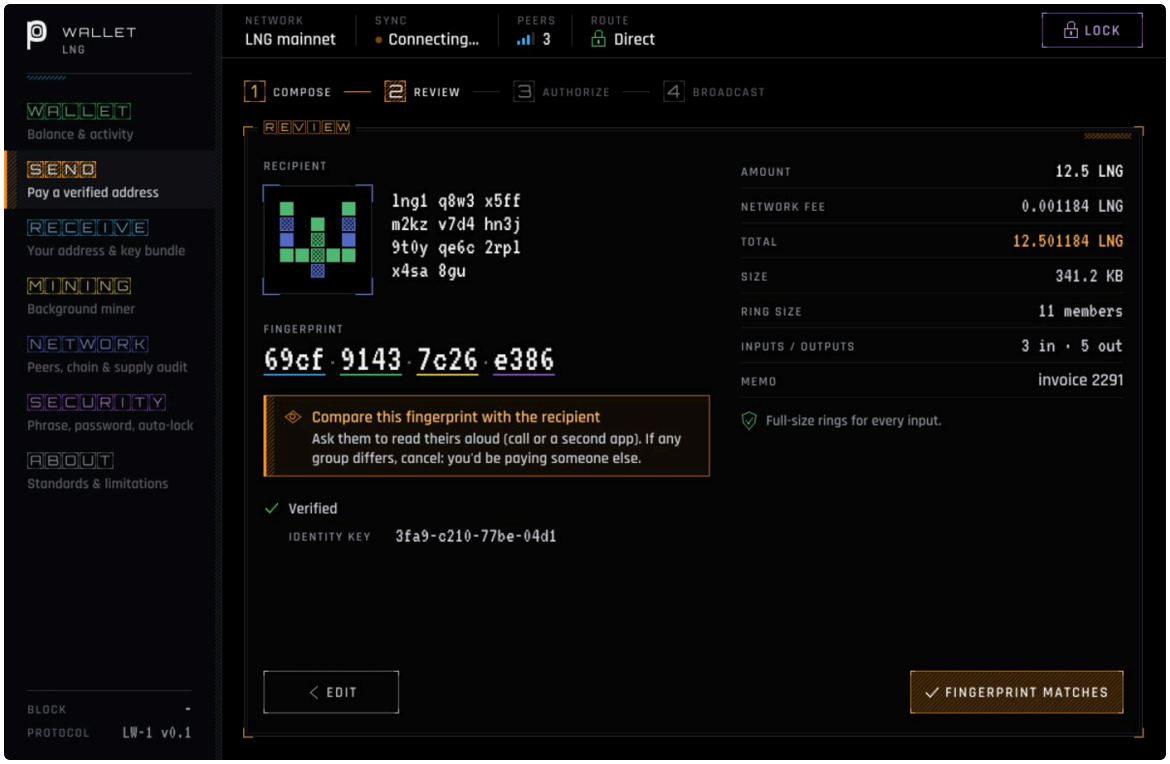


Figure 90.5. The review step of a payment (simulated data): the recipient's identicon and fingerprint, fee, ring size and a prompt to compare the fingerprint out of band.

LingDev

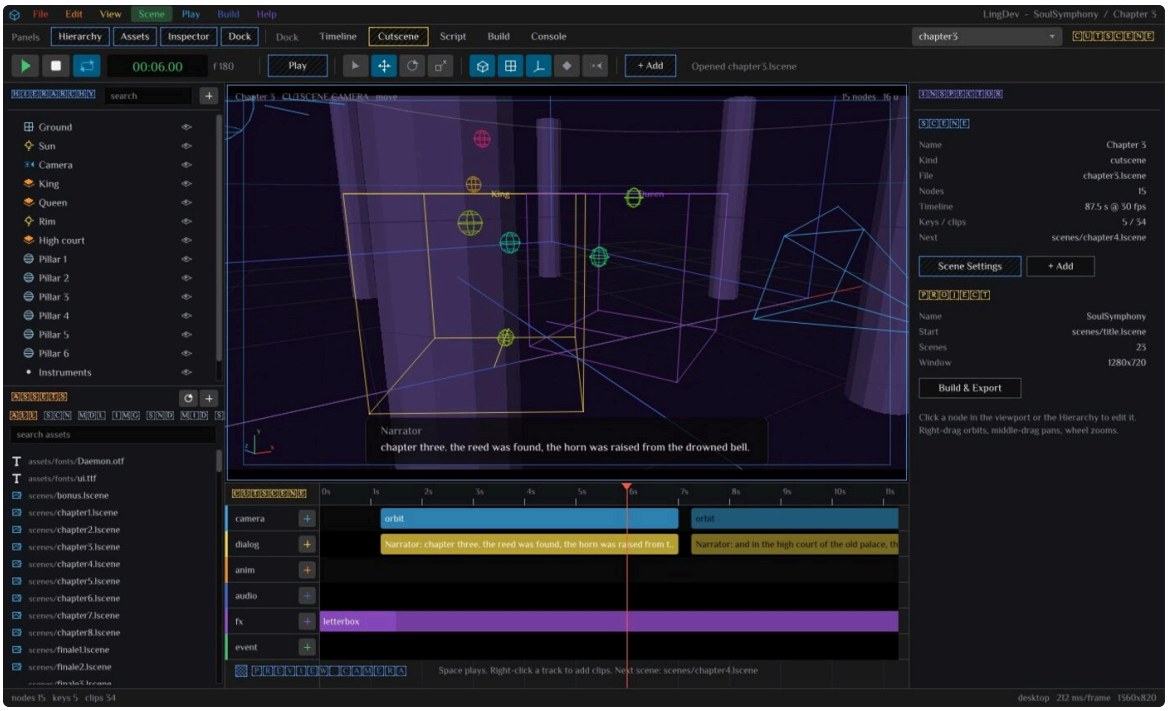


Figure 90.6. LingDev editing a cutscene of the Soul Symphony Ling edition: hierarchy and assets on the left, the 3D viewport with the camera path, the inspector on the right and the cutscene tracks in the dock.

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.

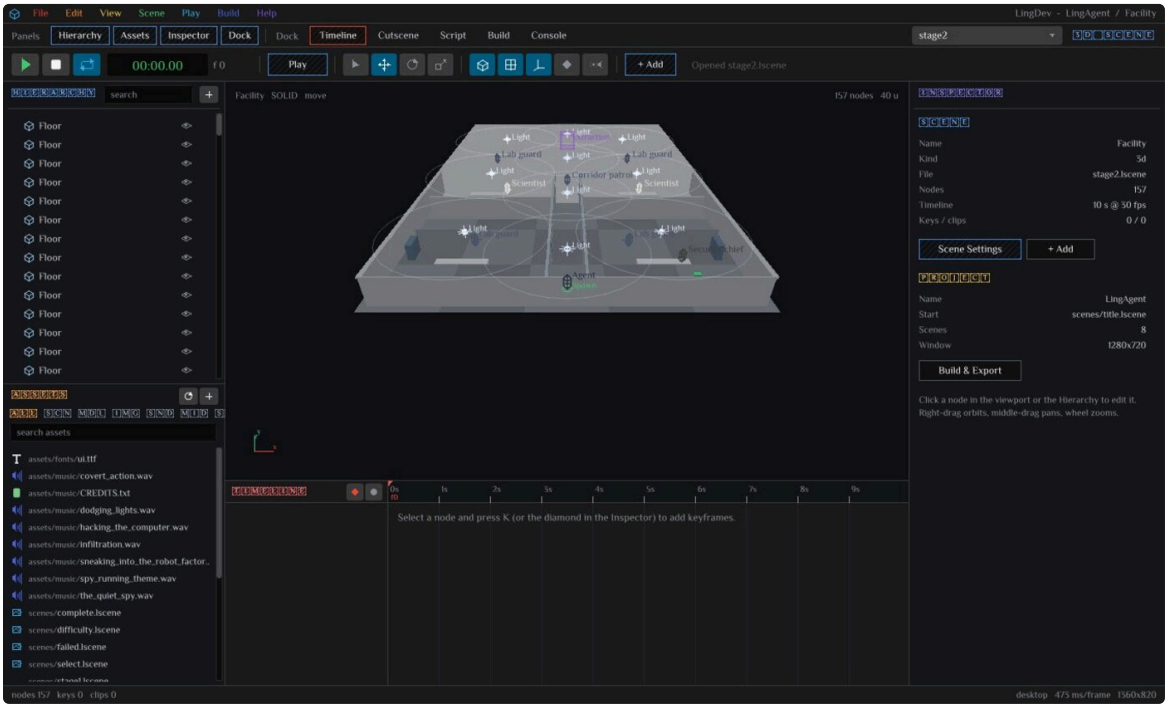


Figure 90.7. The same editor editing a first-person stage: guards, lights and the extraction point in the 3D viewport, drawn by the software renderer in the shared runtime.

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

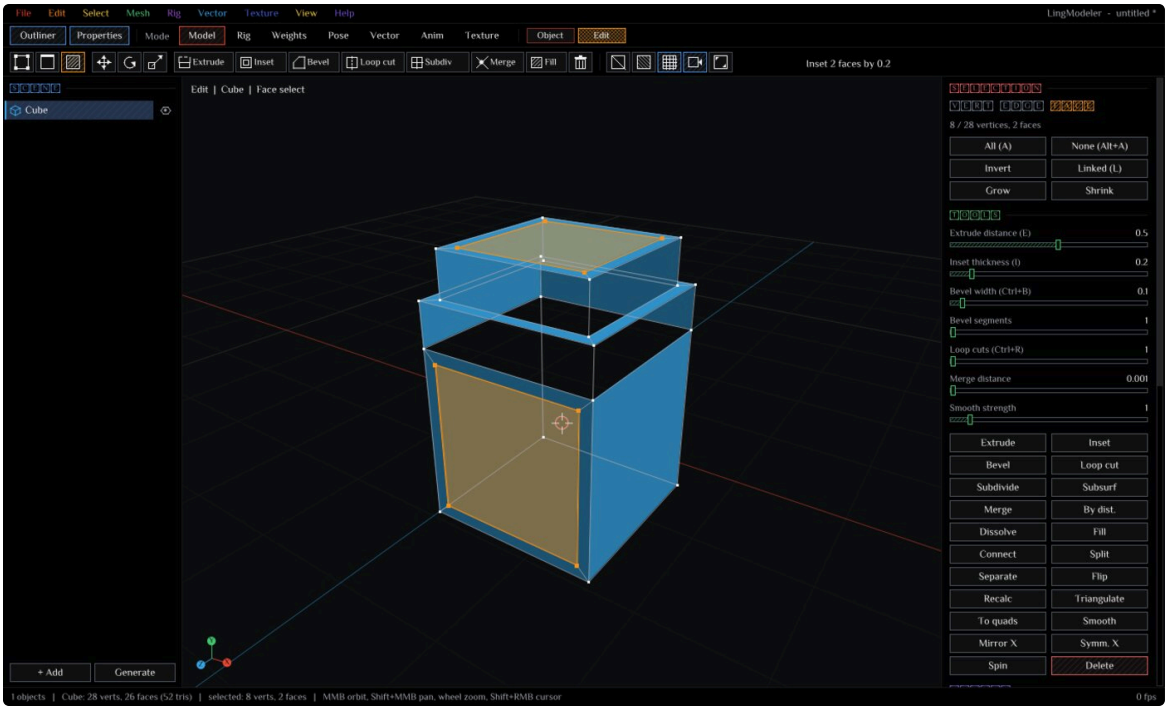


Figure 90.8. Edit mode on a box mesh, with the extrude and inset panels on the right.

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.

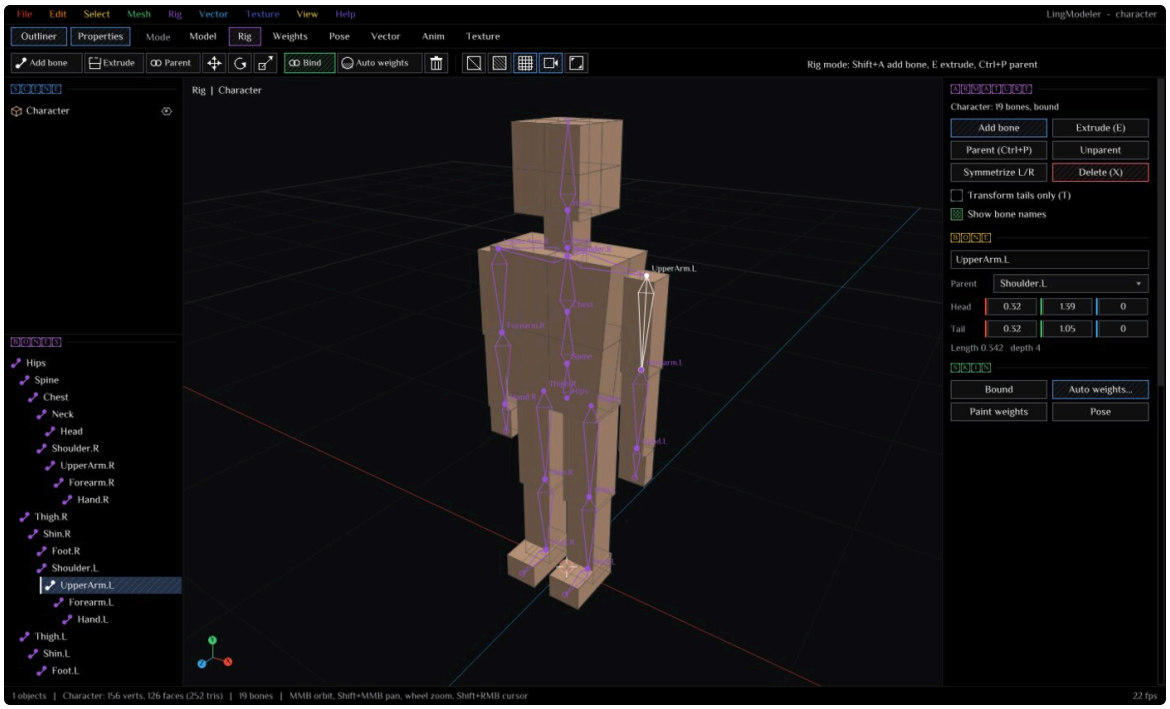


Figure 90.9. Rig mode: a generated character with its 19-bone armature, bone hierarchy on the left and bone properties on the right.

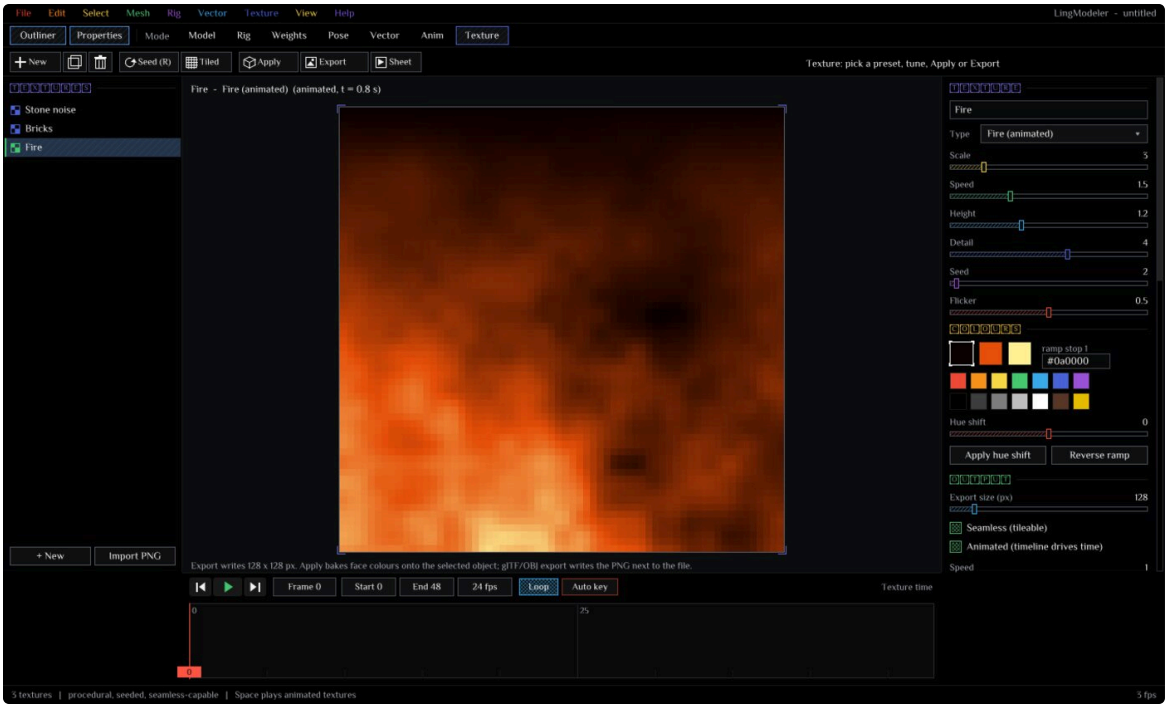


Figure 90.10. The texture lab showing the animated fire generator with its colour ramp and seamless-tiling options.

Horizon

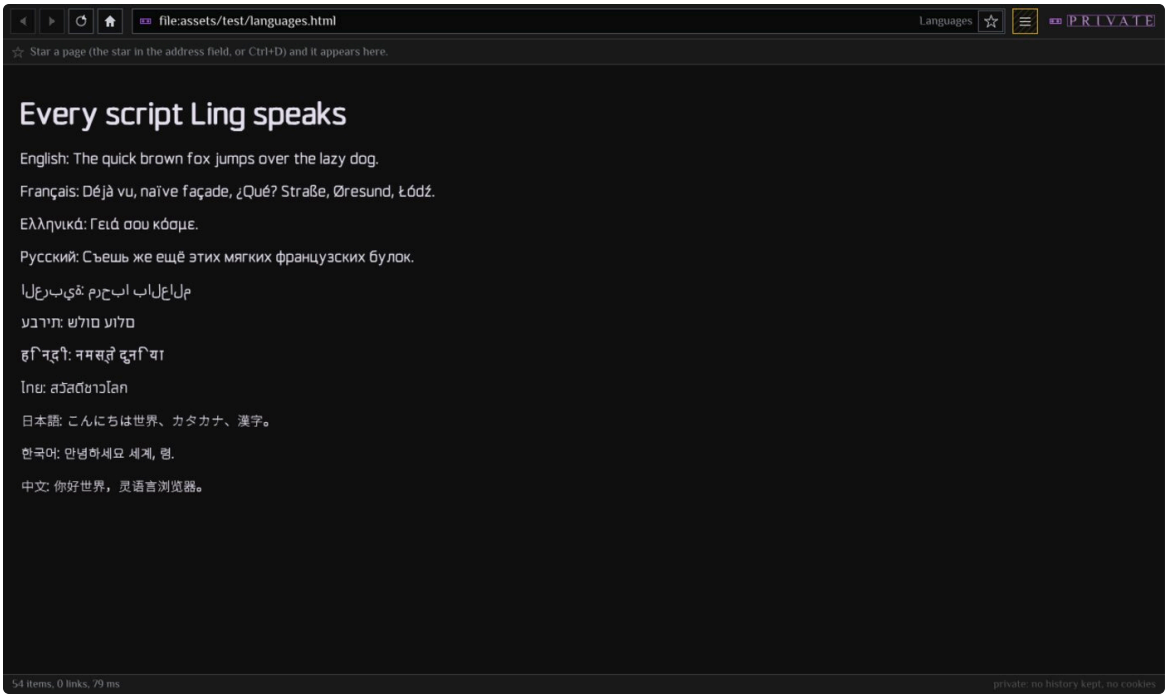


Figure 90.11. Horizon rendering its test page `languages.html`: Latin, Greek, Cyrillic, Arabic, Hebrew, Devanagari, Thai, Japanese, Korean and Chinese, set in its own stroke font.

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

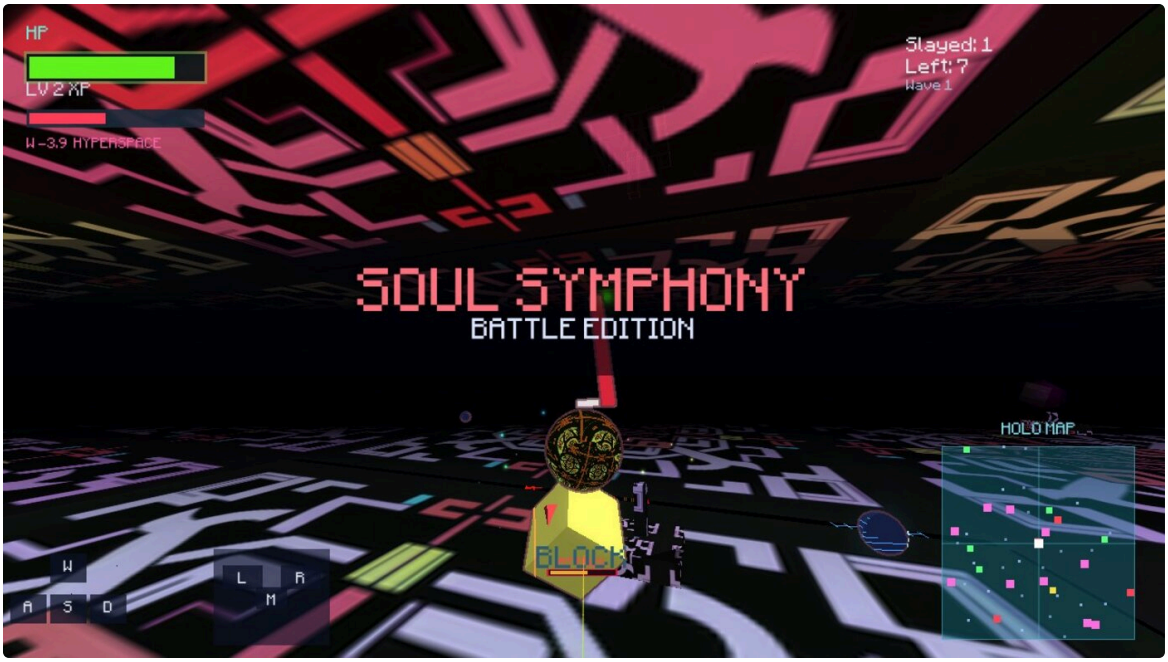


Figure 90.12. Soul Symphony Battle Edition running from a LingDev project: vector lattices, a heads-up display and a holographic map.

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.